TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Simulation of Hybrid Processes

by

R.A. Schouten

Supervisors:
dr.ir. M.A. Reniers
dr.ir. P.J.L. Cuijpers

Eindhoven, August 2005

Master's Thesis for the course "Technische Informatica, 5-jarig (TI5)".
R.A. Schouten, ID-number 456927

Contents

1 Introduction **5**
- 1.1 Motivation and background . . . 5
- 1.2 Related work . . . 5
- 1.3 Development process . . . 5
- 1.4 Structure of this thesis . . . 6

2 HyPA semantics and axioms **7**
- 2.1 Syntax . . . 7
  - 2.1.1 Formal semantics . . . 10
- 2.2 Algebraic reasoning in HyPA . . . 14
  - 2.2.1 Robust bisimilarity of processes . . . 14
  - 2.2.2 Axiomatization . . . 15
  - 2.2.3 Recursion principles . . . 16

3 Requirements **21**
- 3.1 Overview . . . 21
- 3.2 Requirements . . . 21
  - 3.2.1 HyPA input . . . 21
  - 3.2.2 Clause formalisms . . . 23
  - 3.2.3 Visualization . . . 25
  - 3.2.4 Simulation . . . 25
  - 3.2.5 Modularity . . . 29
- 3.3 Requirements sorted by priority . . . 31

4 Analysis **33**
- 4.1 First transitions . . . 33
  - 4.1.1 First transitions function . . . 34
  - 4.1.2 Simulating other classes of HyPA models . . . 36
  - 4.1.3 Optimizations . . . 40
- 4.2 Additional HyPA operators . . . 41
  - 4.2.1 Action renaming . . . 41
  - 4.2.2 Model variable renaming . . . 42
  - 4.2.3 First transitions function . . . 42
  - 4.2.4 Model variable abstraction . . . 43

5 Design **47**
- 5.1 Architecture . . . 47
  - 5.1.1 Rough division . . . 47
  - 5.1.2 User interface . . . 49
  - 5.1.3 Interface . . . 52
  - 5.1.4 Engine . . . 60
- 5.2 Input file format . . . 63

5.3 Clause formalisms . . . 67
5.3.1 Formalism number one . . . 69
5.3.2 Mathematica clause formalism . . . 69
5.3.3 Symbolic Mathematica clause formalism . . . 71

6 Implementation **73**
6.1 Programming language . . . 73
6.2 Coding style . . . 73
6.3 Parsing . . . 73
6.4 Plotting . . . 73

7 Cases **75**
7.1 Steam boiler . . . 75
7.2 Thermostat . . . 78
7.3 Bouncing ball . . . 78
7.4 Billiards . . . 78
7.5 Simple test . . . 81
7.6 Half wave rectifier . . . 83

8 Conclusions **89**

Bibliography **92**

A Mathematical definitions **95**

B Input restrictions **97**

C Proofs for $F_E(t)$ **99**
C.1 $F_E(t)$ is well-defined for substitutably guarded models . . . 99
C.2 $F_E(t)$ is sound . . . 100
C.3 $F_E(t)$ is complete for substitutably guarded models . . . 103
C.4 Renaming operators . . . 106

D Rewriting flows **109**

E Screenshots **111**

F User manual **115**
F.1 Installation . . . 115
F.2 Running . . . 115
F.3 User interface . . . 115
F.4 Loading a HyPA model . . . 118
F.5 Step-wise simulation . . . 118
F.5.1 Solving transitions . . . 118
F.5.2 Choosing a solution . . . 119
F.6 Random simulation . . . 119
F.7 Visualizations . . . 119
F.7.1 Action viewer . . . 120
F.7.2 Plots . . . 120
F.8 Settings . . . 122
F.9 Building . . . 123

G Input file format grammar **125**
G.1 JFlex input . . . 125
G.2 JavaCUP input . . . 128

Chapter 1

Introduction

1.1 Motivation and background

HyPA [6, 7] is a relatively new process algebra that allows for the description of processes that have continuous as well as discrete behavior. The goal of the language is the analysis of such hybrid systems (see [6, section 1.3]). Among the tools for such analysis, a simulator is thought to be useful. It provides the modeler with a first (visual) indication that the model indeed describes the desired process, before more time-consuming analysis tools are applied.

1.2 Related work

There already is a linearization tool for HyPA, described in [18]. It converts a large class of HyPA specifications to equivalent simpler specifications that use a subset of the HyPA operators. This tool might provide a starting point for a simulator: from the resulting form, the possible first transitions are relatively easy to derive. However, in section 4.1 we explain why we do not take this approach.

Hybrid χ [14] is another hybrid formalism developed at the TU/e. A simulator is being constructed simultaneously with ours.

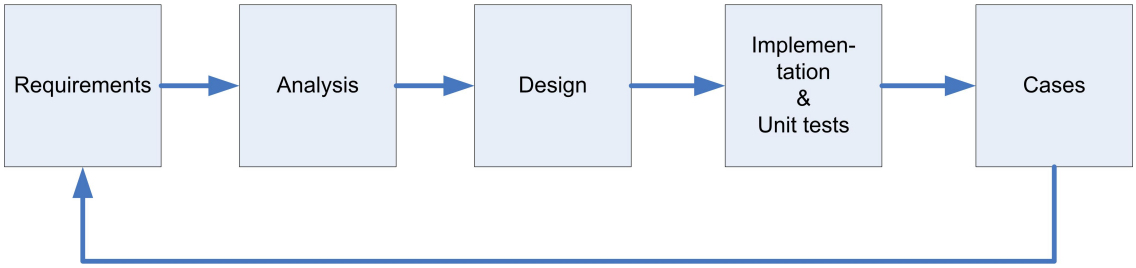
There are other hybrid process simulators like cHARON [1] and HyVisual [9]. cHARON is a hybrid process language that comes with a simulator and a visual editor. HyVisual is a visual modeling language with a simulator. For visualization of the values of model variables, both simulators use the PtPlot Java library [13] that we used in our prototypes as well.

1.3 Development process

Since this is a relatively small project with only one programmer, not much emphasis is placed on following a specific software development standard. The development process is based on an iterative waterfall model (figure 1.1). There is a "requirements" phase during which user requirements are collected. In the "analysis" phase, miscellaneous areas of the problem are analysed. The "design" phase captures the architectural design of the program. There is no "detailed design" phase. Detailed design is provided by structured comments in the code, from which HTML-documentation can be generated. In the "implementation" phase, elements from the eXtreme Programming paradigm [3] are used: programming feature-by-feature, and, for the more complex parts of the program, unit testing. In the "cases" phase, some models are simulated to serve as an integration test. A "maintenance" stage is not provided for. It is unlikely that the tool will gain widespread use, or that there will be any human resources to provide maintenance for the tool.

All phases together form one iteration. The simulator is developed in several iterations. In the

Figure 1.1: Development process phases



successive iterations, emphasis shifts from the first phases to the final ones. The first iterations are primarily for the collection of requirements. After establishing requirements number 1 through 22 (section 3), prototypes of the user interface are produced. Screenshots of these prototypes can be found in Figures E.1 and E.2. The prototypes show the different visualizations of model behavior, and indicate the way in which the simulator will be controlled by the user. Evaluation of the prototypes leads to requirements 23 through 47. After two user interface prototypes, a new iteration is started which produces a first simulator version. Successive iterations provide simulators that implement more and more of the requirements.

1.4 Structure of this thesis

Chapter 2 gives an introduction to the HyPA language. After that, the phases of the development process ("requirements", "analysis", ...) all have their own chapter. Finally, conclusions are presented in chapter 8.

Chapter 2

HyPA semantics and axioms

This chapter gives an introduction to HyPA. It is a slightly compressed copy of [6, chapter 3]. First, the syntax is described, followed by the semantics and axioms. The HyPA axioms have been rearranged and numbered for ease of reference in the rest of this document. Finally, definition 2.12 is taken from [18, 19].

2.1 Syntax

The signature of HyPA consists of the following constant and function symbols:

1. deadlock δ,
2. empty process ϵ,
3. discrete actions $a \in \mathcal{A}$,
4. flow clauses $c \in C$,
5. a family of process re-initialization operators $(d \gg _)_{d \in D}$,
6. alternative composition $_ \oplus _$,
7. sequential composition $_ \odot _$,
8. disrupt $_ \blacktriangleright _$ and left-disrupt $_ \rhd _$,
9. parallel composition $_ \parallel _$, left-parallel composition $_ \between _$, and forced-synchronization $_ \mid _$,
10. a family of encapsulation operators $(\partial_H(_))_{H \subseteq \mathcal{A}}$.

Deadlock and the empty process

The atomic process terms δ (called *deadlock*) and ϵ (called *empty process*) are used to model a process that shows no behavior at all (i.e. that has inadvertently stopped functioning) and a (successfully) terminating process, respectively.

Discrete actions

The atomic *discrete actions* are used to model discrete, computational behavior. The set $\mathcal{A}$ of discrete actions is considered a parameter of the theory and can be instantiated at will by the user of our hybrid process algebra.

Flow clauses

Flow clauses are used to model continuous physical behavior. Traditionally, in systems theory, several different formalisms are used for the description of continuous behavior, and often the modeling or analysis question determines which formalism is to be used.

Common to all approaches, is that continuous behavior is described by some sort of predicate on the flow of values of model variables $\mathcal{V}_\mathrm{m}$ through time. Formally, we write $\mathcal{V} = \bigcup_{x \in \mathcal{V}_\mathrm{m}} \mathcal{V}(x)$ for the union of all model variable domains, and $\mathit{Val} = \mathcal{V}_\mathrm{m} \to \mathcal{V}$ for the set of variable valuations. The set of all flows is $\mathcal{F} = \{ f \in T \mapsto \mathit{Val} \mid \mathrm{dom}(f) = [0, t] \text{ for some } t \in T \}$. Note, that these functions all have a closed-interval domain starting in 0. The flows that are described by a flow predicate, are called solutions of that predicate. We consider the set of flow predicates $\mathcal{P}_\mathrm{f}$, the sets $\mathcal{V}_\mathrm{m}$ of model variables and T of time points, and the notion of solution $\models_\mathrm{f} \subseteq \mathcal{F} \times \mathcal{P}_\mathrm{f}$, that defines which flows are considered solutions of a flow predicate, parameters of our theory. This means they can be instantiated by the modeler, depending on the specific modeling or analysis problem. The theory we present in this chapter, is largely independent of that choice, except that we assume the existence of a flow predicate $\mathit{false} \in \mathcal{P}_\mathrm{f}$ that satisfies no flow from the set $\mathcal{F}$.

An atomic flow clause, finally, is a pair $(V \,|\, P_\mathrm{f})$ of a set of model variables $V \subseteq \mathcal{V}_\mathrm{m}$, signifying which variables are not allowed to jump at the beginning of a flow, and a flow predicate $P_\mathrm{f} \in \mathcal{P}_\mathrm{f}$ modeling continuous, never terminating, physical behavior. The set of all flow clauses is denoted C. We usually leave out the brackets for V, and even omit it (and the '|' delimiter) if it is empty. Furthermore, the set C is closed under conjunction ($\wedge$) of flow clauses, and using the assumption that there is a flow predicate *false*, which is never satisfied, there is also a flow clause (*false*), which is the system theoretic equivalent of deadlock δ.

Re-initializations

As with continuous physical behavior, there are several formalisms in systems theory that deal with discontinuous physical behavior, and, again, the modeling or analysis question determines which formalism is to be used in a specific situation. In general, one may say that discontinuous behavior is described through predicates about the re-initialization (or discontinuity, or change) of variables. As an example of such a predicate, consider a difference equation, $x^+ = f(x^-, u^-)$, which denotes that the value of x is reassigned to $f(x^-, u^-)$, based on the previous values of x and u.

Re-initialization predicates describe a set of re-initializations, which are pairs of valuations representing the values of the model variables prior to and immediately after the re-initialization. Such re-initializations are called solutions of the re-initialization predicate. The set of all re-initializations $\mathit{Val} \times \mathit{Val}$ is denoted $\mathcal{R}$. As before, the set of re-initialization predicates $\mathcal{P}_\mathrm{r}$ and the notion of solution $\models_\mathrm{r} \subseteq \mathcal{R} \times \mathcal{P}_\mathrm{r}$, that defines which re-initializations are considered solutions of a re-initialization predicate, are considered parameters of the theory. We assume the existence of re-initialization predicates $\mathit{true}, \mathit{false} \in \mathcal{P}_\mathrm{r}$ that satisfy any re-initialization, and no re-initialization from the set $\mathcal{R}$, respectively.

A process re-initialization $d \gg p$ models the behavior of p where the model variables are submitted to a discontinuous change as specified by the re-initialization clause d. A re-initialization clause is a pair $[V \,|\, P_\mathrm{r}]$ of a set of model variables $V \subseteq \mathcal{V}_\mathrm{m}$ and a re-initialization predicate P_r. The set V models which variables are allowed to change. Note that this is precisely opposite to flow clauses, where V denotes those variables that do not change. The set of all re-initialization clauses is denoted D. The set D is closed under conjunction ($\wedge$), disjunction ($\vee$), and concatenation ($\sim$) of re-initialization clauses. Also, there is a satisfiability operator ($d^?$) on clauses $d \in D$, which does not re-initialize the values of a model variable, but only executes the re-initialized process, if d can be satisfied in some way. And finally, there is a re-initialization clause (c_{jmp}) derived from a flow clause $c \in C$, which executes the same discontinuities that are allowed initially by the flow clause. These last two operators turn out to be especially useful when calculating with process terms. Using the assumption that there are re-initialization predicates *false* and *true*, we find the process re-initialization $[\mathit{false}] \gg p$, executing no behavior since there is no re-initialization satisfying

false, the process re-initialization $[true] \gg p$, executing exactly the behavior of p, since none of the variables is allowed to change, and the process re-initialization $[\mathcal{V}_{\mathrm{m}}|\, true] \gg p$, executing p after an arbitrary re-initialization.

Alternative and sequential composition

The alternative composition $p \oplus q$ models a (non-deterministic) choice between the processes p and q. The sequential composition $p \odot q$ models a sequential execution of processes p and q. The process q is executed after (successful) termination of the process p.

We use the notations $\oplus$ and $\odot$ for alternative and sequential composition, rather than the usual $+$ and $\cdot$, to avoid confusion with the notation used frequently in the description of flow and re-initialization predicates for addition and multiplication. We realize that this might distract people in the field of process algebra, yet chose to adapt the process algebraic notation rather than the notation adopted from system theory, simply because the latter has been in use for a longer time already. Overloading the operators is also an option, since it is always clear from the context whether for example addition or choice is intended. When studying HyPA as a new process algebra, as is done in this thesis, overloading is probably to be preferred indeed, as it hardly hampers the search for process algebraic properties. However, when studying hybrid models in HyPA, and performing analysis using axioms from both process algebra and system theory in the same proofs, the overloading becomes more of a burden. Furthermore, when presenting these models to other hybrid researchers who are often not familiar with process algebra at all, this effect is even stronger.

Disrupt

The disrupt $p \blacktriangleright q$ models a kind of sequential composition where the process q may take over execution from process p at any moment, without waiting for its termination. This composition is invaluable when modeling two flow clauses executing one after the other, since the behavior of flow clauses is ongoing, and never terminates. The left-disrupt is mainly needed for calculation and axiomatization purposes, rather than for modeling purposes. For example, it occurs often when we attempt to eliminate the parallel composition from a process term through axiomatic reasoning, as described in section 2.2. The left-disrupt $p \rhd q$ first executes a part of the process p and then behaves as a normal disrupt.

Parallel composition

The parallel composition $p \parallel q$ models concurrent execution of p and q. The intuition behind this concurrent execution is that discrete actions are executed in an interleaving manner, with the possibility of synchronization, while flow clauses are forced to synchronize, and can only synchronize if they accept the same solutions. The synchronization of actions takes place using a (partial, commutative, and associative) communication function $\gamma \in \mathcal{A} \times \mathcal{A} \mapsto \mathcal{A}$. For example, if the actions a and a' synchronize, the resulting action is $a'' = a\gamma a'$. Actions cannot synchronize with flow clauses, and in a parallel composition between those, the action executes first. This communication function is considered a parameter of the theory.

As with the left-disrupt, the operators left-parallel composition and forced-synchronization are mainly introduced for calculation purposes. The left-parallel composition $p \mathbin{\lfloor\!\lfloor} q$ models that either p performs a discrete action first, and then behaves as a normal parallel composition with q, or p cannot perform such an action, and the process deadlocks. The forced-synchronization $p \mid q$ models how the first behavior (either a discrete action or a part of a flow) of p and q is synchronized, after which they behave as in a normal parallel composition. If synchronization is not possible, then the forced-synchronization deadlocks.

Encapsulation

Encapsulation $\partial_H(p)$ models that certain discrete actions (from the set $H \subseteq \mathcal{A}$) are blocked during the execution of the process p. This operator is often used in combination with the parallel composition to model that synchronization between discrete actions is enforced.

From the signature of HyPA, terms can be constructed using variables from a given set of process variables $\mathcal{V}_\mathrm{p}$ (with $\mathcal{V}_\mathrm{p} \cap \mathcal{V}_\mathrm{m} = \emptyset$), as usual. In this thesis, the set of all such terms is denoted $\mathcal{T}(\mathcal{V}_\mathrm{p})$ and these are referred to as terms or open terms. Terms in which no process variables occur are called closed terms. The set of all closed terms is denoted $\mathcal{T}$.

Finally, all the processes should be interpreted in the light of a set E of recursive definitions, called *recursive specification*, of the form $X : p$, where X is a process variable and p is a term. We denote the set of all process variables that occur in the left-hand side of a recursive definition from E by $\mathcal{V}_\mathrm{r}$ ($\mathcal{V}_\mathrm{r} \subseteq \mathcal{V}_\mathrm{p}$) and call these variables recursion variables. The set $\mathcal{T}(\mathcal{V}_\mathrm{r})$ denotes the set of all terms in which only recursion variables are used. Such elements are referred to as process terms. We only allow recursive definitions $X : p$ where the term p is a process term. Outside the recursive specification, recursion variables are treated as constants of the theory. Recursion is a powerful way to model repetition in a process.

The binding order of the operators of HyPA is as follows: $\odot$, $\blacktriangleright$, $\rhd$, $d \gg$, $\|$, $\between$, $|$, $\oplus$, where alternative composition binds weakest, and sequential composition binds strongest. With encapsulation ($\partial_H(_)$), brackets are always used. As an example, a term $d \gg a \odot b \oplus c \| c'$ should be read as $(d \gg (a \odot b)) \oplus (c \| c')$.

2.1.1 Formal semantics

In this section, we give a formal semantics to the syntax defined in the previous section, by constructing a hybrid labeled transition system, for each process term and each possible valuation of the model variables. The hybrid transition systems we use here, have two different kinds of transitions: one associated with computational behavior (i.e. discrete actions), and the other associated with physical behavior (i.e. flow clauses). There is also a termination predicate ($\checkmark$) on states, indicating successful termination of a computation. Furthermore, we have adapted the notation in this chapter, to the notation that is more commonly used in the development of process algebras. The system structure is indicated by two separate transition relations $\rightsquigarrow \subseteq X \times \Sigma \times X$ and $\mapsto \subseteq X \times A \times X$.

Definition 2.1 (Hybrid Transition System)
A *hybrid transition system* is a tuple $\langle X, A, \Sigma, \mapsto, \rightsquigarrow, \checkmark \rangle$, consisting of a *state space* X, a *set of action labels* A, a *set of flow labels* Σ, and *transition relations* $\mapsto \subseteq X \times A \times X$ and $\rightsquigarrow \subseteq X \times \Sigma \times X$. Lastly, there is a *termination predicate* $\checkmark \subseteq X$.

For the semantical hybrid transition systems that are associated with HyPA terms, the state space is formed by pairs of process terms and valuations of the model variables, i.e. $X = \mathcal{T}(\mathcal{V}_\mathrm{r}) \times \mathit{Val}$. The set of action labels is formed by pairs of actions and valuations, i.e. $A = \mathcal{A} \times \mathit{Val}$, and the set of flow labels is formed by the set of flows, i.e. $\Sigma = \mathcal{F}$. Recall that the elements $f \in \mathcal{F}$ have a closed-interval domain, possibly a singleton, starting in 0.

Let $x, x' \in X$ be two states. We use the notation $\langle x \rangle \overset{a}{\mapsto} \langle x' \rangle$ for a transition $(x, a, x') \in \mapsto$ with $a \in A$. Similarly, we use $\langle x \rangle \overset{\sigma}{\rightsquigarrow} \langle x' \rangle$ for a transition $(x, \sigma, x') \in \rightsquigarrow$ with $\sigma \in \Sigma$. For arbitrary transitions, we use $\langle x \rangle \overset{l}{\rightarrow} \langle x' \rangle$ instead of $(x, l, x') \in \mapsto \cup \rightsquigarrow$, with $l \in A \cup \Sigma$. Finally, termination is denoted $\langle x \rangle \checkmark$ instead of $x \in \checkmark$.

Before we turn to the actual definition of the semantics of HyPA in terms of hybrid transition systems, a notion of solution for flow clauses and re-initialization clauses is needed for the definition of the semantics of these atoms of the algebra. These notions are obtained by lifting the notion of solution of flow predicates and re-initialization predicates, while taking into account the influence of the variable set V.

A flow clause $(V|\, P_{\mathrm{f}})$ changes the valuation of the model variables according to the possible solutions of its flow predicate P_{f}. An initial jump in the value of a variable x is allowed in HyPA when $x \notin V$. Furthermore, discontinuous and non-differentiable flows of x may be allowed, if such solutions exists for the type of flow predicate that is used. The concept of solution of a flow clause, is lifted from the notion of solutions of its flow predicate as follows.

Definition 2.2 (Solution of a flow clause)
. *A pair $(\nu, \sigma) \in \mathit{Val} \times \mathcal{F}$, is defined to be a* solution *of a flow clause $c \in C$, denoted $(\nu, \sigma) \models c$, as follows:*

- $(\nu, \sigma) \models (V|\, P_{\mathrm{f}})$ *if* $\sigma \models_{\mathrm{f}} P_{\mathrm{f}}$, *and for all* $x \in V$ *we find* $\nu(x) = \sigma(0)(x)$;
- $(\nu, \sigma) \models c \wedge c'$ *if* $(\nu, \sigma) \models c$ *and* $(\nu, \sigma) \models c'$.

Clearly, the flow clause (*false*) has no solutions, as the flow predicate *false* has no solutions.

A re-initialization clause $[V|\, P_{\mathrm{r}}]$ changes the valuation of the model variables according to the possible solutions of its re-initialization predicate P_{r}. The set V indicates the variables that are allowed to change their value. Whenever $x \notin V$, the variable x is fixed. Note that this is precisely opposite to the use of V in flow clauses. We define the solutions of a re-initialization clause in terms of the solutions of a re-initialization predicate as follows.

Definition 2.3 (Solution of a re-initialization clause)
A re-initialization
$(\nu, \nu') \in \mathcal{R}$ is defined to be a *solution* of a re-initialization clause $d \in D$, denoted $(\nu, \nu') \models d$, as follows:

- $(\nu, \nu') \models [V|\, P_{\mathrm{r}}]$ if $(\nu, \nu') \models_{\mathrm{r}} P_{\mathrm{r}}$ and for all $x \notin V$ we find $\nu(x) = \nu'(x)$;
- $(\nu, \nu') \models d' \vee d''$ if $(\nu, \nu') \models d'$ or $(\nu, \nu') \models d''$;
- $(\nu, \nu') \models d' \wedge d''$ if $(\nu, \nu') \models d'$ and $(\nu, \nu') \models d''$;
- $(\nu, \nu') \models d' \sim d''$ if there exists $\upsilon \in \mathit{Val}$ with $(\nu, \upsilon) \models d'$ and $(\upsilon, \nu') \models d''$;
- $(\nu, \nu') \models d'^{?}$ if $\nu = \nu'$, and there exists $\upsilon \in \mathit{Val}$ with $(\nu, \upsilon) \models d'$;
- $(\nu, \nu') \models c_{jmp}$ if there exists $\sigma \in \Sigma$ such that $(\nu, \sigma) \models c$ and $\sigma(0) = \nu'$.

If we have two re-initialization clauses $d, d' \in D$, the clause $d \sim d'$ accepts exactly those solutions that are a concatenation of the re-initializations of d and d'. The clause $d^{?}$ does not change the value of any of the variables, it just models the condition under which d has a solution. The clause c_{jmp} imitates the re-initializations performed initially by a flow clause c. Obviously, the re-initialization clause [*false*] has no solutions, while $[\mathcal{V}_{\mathrm{m}}|\, \mathit{true}]$ has every possible re-initialization as a solution. Note, that [*true*] exactly allows all re-initializations that do not change any of the variable valuations.

The semantics of the HyPA constants and function symbols is given in the tables 1–5, using deduction rules in the style of [12]. In these tables p, p', q, q' denote process terms, a, a', a'' denote actions, c denotes a flow clause, d denotes a re-initialization clause, H denotes a set of actions, X denotes a recursion variable, ν, ν', ν'' denote valuations, σ denotes a flow, t denotes a point in time, and l denotes an arbitrary transition label.

In table 2.1, the semantics of the atomic processes, the flow clauses, and the process re-initializations is given. Rule (1) captures our intuition that ϵ is a process that only terminates. Analogously, the fact that there is no rule for δ, expresses that this is indeed a deadlocking process. Rule (2) expresses that discrete actions display their own name, and the valuation of the model

Table 2.1: Operational semantics of HyPA

$$\frac{}{\langle\, \epsilon, \nu \,\rangle \checkmark}(1) \quad \frac{}{\langle\, a, \nu \,\rangle \overset{a,\nu}{\mapsto} \langle\, \epsilon, \nu \,\rangle}(2) \quad \frac{(\nu, \sigma) \models c,\ dom(\sigma) = [0,t]}{\langle\, c, \nu \,\rangle \overset{\sigma}{\rightsquigarrow} \langle\, c, \sigma(t) \,\rangle}(3)$$

$$\frac{(\nu, \nu') \models d,\ \langle\, p, \nu' \,\rangle \checkmark}{\langle\, d \gg p, \nu \,\rangle \checkmark}(4) \quad \frac{(\nu, \nu') \models d,\ \langle\, p, \nu' \,\rangle \overset{l}{\to} \langle\, p', \nu'' \,\rangle}{\langle\, d \gg p, \nu \,\rangle \overset{l}{\to} \langle\, p', \nu'' \,\rangle}(5)$$

Table 2.2: Operational semantics of HyPA, alternative and sequential composition

$$\frac{\langle\, p, \nu \,\rangle \checkmark}{\begin{array}{l}\langle\, p \oplus q, \nu \,\rangle \checkmark \\ \langle\, q \oplus p, \nu \,\rangle \checkmark\end{array}}(6) \quad \frac{\langle\, p, \nu \,\rangle \overset{l}{\to} \langle\, p', \nu' \,\rangle}{\begin{array}{l}\langle\, p \oplus q, \nu \,\rangle \overset{l}{\to} \langle\, p', \nu' \,\rangle \\ \langle\, q \oplus p, \nu \,\rangle \overset{l}{\to} \langle\, p', \nu' \,\rangle\end{array}}(7)$$

$$\frac{\langle\, p, \nu \,\rangle \checkmark,\ \langle\, q, \nu \,\rangle \checkmark}{\langle\, p \odot q, \nu \,\rangle \checkmark}(8) \quad \frac{\langle\, p, \nu \,\rangle \overset{l}{\to} \langle\, p', \nu' \,\rangle}{\langle\, p \odot q, \nu \,\rangle \overset{l}{\to} \langle\, p' \odot q, \nu' \,\rangle}(9)$$

$$\frac{\langle\, p, \nu \,\rangle \checkmark,\ \langle\, q, \nu \,\rangle \overset{l}{\to} \langle\, q', \nu' \,\rangle}{\langle\, p \odot q, \nu \,\rangle \overset{l}{\to} \langle\, q', \nu' \,\rangle}(10)$$

variables on the transition label, but do not change this valuation. Changes in the valuation can only be caused by flow clauses and re-initialization clauses, as defined by rules (3) to (5).

The semantics of the other operators is defined in tables 2.2, 2.3, 2.4, and 2.5.

Table 2.3: Operational semantics of HyPA, disrupt

$$\frac{\langle\, p, \nu \,\rangle \checkmark}{\begin{array}{l}\langle\, p \blacktriangleright q, \nu \,\rangle \checkmark \\ \langle\, p \rhd q, \nu \,\rangle \checkmark\end{array}}(11) \quad \frac{\langle\, p, \nu \,\rangle \overset{l}{\to} \langle\, p', \nu' \,\rangle}{\begin{array}{l}\langle\, p \blacktriangleright q, \nu \,\rangle \overset{l}{\to} \langle\, p' \blacktriangleright q, \nu' \,\rangle \\ \langle\, p \rhd q, \nu \,\rangle \overset{l}{\to} \langle\, p' \blacktriangleright q, \nu' \,\rangle\end{array}}(12)$$

$$\frac{\langle\, q, \nu \,\rangle \checkmark}{\langle\, p \blacktriangleright q, \nu \,\rangle \checkmark}(13) \quad \frac{\langle\, q, \nu \,\rangle \overset{l}{\to} \langle\, q', \nu' \,\rangle}{\langle\, p \blacktriangleright q, \nu \,\rangle \overset{l}{\to} \langle\, q', \nu' \,\rangle}(14)$$

Rules (11) to (14) define the semantics of the disrupt operator and the left-disrupt operator. If we compare these rules to the rules for sequential composition, we see that the main difference, is the way in which termination is handled. Firstly, in a composition $p \blacktriangleright q$, the process q may start execution without p terminating. Secondly, if the process p terminates, the process $p \blacktriangleright q$ may also terminate regardless of the behavior of q.

Rules (15) to (19) define the semantics of the parallel composition, and in these rules the difference between action transitions and flow transitions is most prominent. Discrete actions that are placed in parallel are interleaved, but can also synchronize using a (partial, commutative, and

Table 2.4: Operational semantics of HyPA, parallel composition

$$\frac{\langle p,\nu\rangle\checkmark,\langle q,\nu\rangle\checkmark}{\begin{array}{c}\langle p\parallel q,\nu\rangle\checkmark\\ \langle p\mid q,\nu\rangle\checkmark\end{array}}(15)\qquad \frac{\langle p,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p',\nu'\rangle,\ \langle q,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle q',\nu'\rangle}{\begin{array}{c}\langle p\parallel q,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p'\parallel q',\nu'\rangle\\ \langle p\mid q,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p'\parallel q',\nu'\rangle\end{array}}(16)$$

$$\frac{\langle p,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p',\nu'\rangle,\ \langle q,\nu\rangle\checkmark}{\begin{array}{c}\langle p\parallel q,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p',\nu'\rangle\\ \langle q\parallel p,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p',\nu'\rangle\\ \langle p\mid q,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p',\nu'\rangle\\ \langle q\mid p,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p',\nu'\rangle\end{array}}(17)\qquad \frac{\langle p,\nu\rangle\overset{a,\nu'}{\mapsto}\langle p',\nu''\rangle}{\begin{array}{c}\langle p\parallel q,\nu\rangle\overset{a,\nu'}{\mapsto}\langle p'\parallel q,\nu''\rangle\\ \langle q\parallel p,\nu\rangle\overset{a,\nu'}{\mapsto}\langle q\parallel p',\nu''\rangle\\ \langle p\between q,\nu\rangle\overset{a,\nu'}{\mapsto}\langle p'\parallel q,\nu''\rangle\end{array}}(18)$$

$$\frac{\langle p,\nu\rangle\overset{a,\nu'}{\mapsto}\langle p',\nu''\rangle,\ \langle q,\nu\rangle\overset{a',\nu'}{\mapsto}\langle q',\nu''\rangle,\ a''=a\,\gamma\,a'}{\begin{array}{c}\langle p\parallel q,\nu\rangle\overset{a'',\nu'}{\mapsto}\langle p'\parallel q',\nu''\rangle\\ \langle p\mid q,\nu\rangle\overset{a'',\nu'}{\mapsto}\langle p'\parallel q',\nu''\rangle\end{array}}(19)$$

associative) communication function $\gamma\in\mathcal{A}\times\mathcal{A}\mapsto\mathcal{A}$. If a discrete action a communicates with an action a' (this is the case if $a\gamma a'$ is defined), the result is an action $a''=a\gamma a'$. If flow clauses are placed in parallel, they always synchronize their behavior such that, intuitively, the flows that are possible in a parallel composition are a solution of both clauses.

Encapsulation, as defined by rules (20) to (22), only influences action transitions. This is not surprising, since, as mentioned before, the $\partial_H(_)$ operator is originally intended to model enforced synchronization in a parallel composition. Parallel composition, in general, may lead to interleaving actions and synchronized actions. The encapsulation operator is then used to block the interleaving actions. Flow transitions are already synchronized in the parallel composition, so there is no need for encapsulation of those.

Table 2.5: Operational semantics of HyPA, encapsulation and recursion

$$\frac{\langle p,\nu\rangle\overset{a,\nu'}{\mapsto}\langle p',\nu''\rangle,\ a\notin H}{\langle\partial_H(p),\nu\rangle\overset{a,\nu'}{\mapsto}\langle\partial_H(p'),\nu''\rangle}(20)$$

$$\frac{\langle p,\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle p',\nu'\rangle}{\langle\partial_H(p),\nu\rangle\overset{\sigma}{\rightsquigarrow}\langle\partial_H(p'),\nu'\rangle}(21)\qquad \frac{\langle p,\nu\rangle\checkmark}{\langle\partial_H(p),\nu\rangle\checkmark}(22)$$

$$\frac{\langle p,\nu\rangle\checkmark}{\langle X,\nu\rangle\checkmark}(23)\quad X:p\in E\qquad \frac{\langle p,\nu\rangle\overset{l}{\rightarrow}\langle p',\nu'\rangle}{\langle X,\nu\rangle\overset{l}{\rightarrow}\langle p',\nu'\rangle}(24)\quad X:p\in E$$

Rules (23) and (24) model recursion. For a recursive definition $X:p$, a transition for the variable X is possible, if it can be deduced from the semantical rules for the process term p.

2.2 Algebraic reasoning in HyPA

The strength of the field of process algebra, lies in its ability to use equational reasoning for the analysis of transition systems, or, more precisely, for the analysis of equivalence classes of transition systems, called processes. In this section, we show that this equational reasoning is also possible in HyPA.

We start out by defining a notion of bisimilarity on process terms, reflecting equivalence of the underlying hybrid transition systems. Then, we study properties of this equivalence, and capture those properties in a set of *derivation rules* and a set of *axioms* on the algebra of process terms. Together with a *principle for guarded recursion*, this forms a proof system in which every derived equality on process terms represents equality of the underlying hybrid transition systems. In other words, process terms that are derivably equal, describe transition systems in the same equivalence class, and hence describe the same process.

In the first part of this section, we define the notion of robust bisimilarity. In the second part, we give a formal axiomatization of this notion, and we treat the intuition behind the axioms, and the insights they provide us with. In the third part, we discuss a specification principle that is used for reasoning about recursion.

2.2.1 Robust bisimilarity of processes

The hybrid transition systems that are used to give a semantics of HyPA, contain an additional termination structure. Therefore, we start out by adapting the notion of bisimilarity on hybrid transition systems as follows.

Definition 2.4 (Bisimilarity on hybrid transition systems)
Given, a hybrid transition system $\langle X, A, \Sigma, \mapsto, \rightsquigarrow, \checkmark \rangle$ (see [6, definition 1 and 2]), a relation $R \subseteq X \times X$ is a *bisimulation* relation if

- for all $x, y \in X$ such that $x \,R\, y$, we find $\langle x \rangle \checkmark$ implies $\langle y \rangle \checkmark$;
- for all $x, y \in X$ such that $x \,R\, y$, we find $\langle y \rangle \checkmark$ implies $\langle x \rangle \checkmark$;
- for all $x, x', y \in X$ such that $x \,R\, y$ and $l \in A \cup \Sigma$, we find $\langle x \rangle \xrightarrow{l} \langle x' \rangle$ implies there exists y' such that $\langle y \rangle \xrightarrow{l} \langle y' \rangle$ and $x' \,R\, y'$;
- for all $x, y, y' \in X$ such that $x \,R\, y$ and $l \in A \cup \Sigma$, we find $\langle y \rangle \xrightarrow{l} \langle y' \rangle$ implies there exists x' such that $\langle x \rangle \xrightarrow{l} \langle x' \rangle$ and $x' \,R\, y'$.

Two states $x, y \in X$ are *bisimilar*, notation $x \leftrightarroweq y$, if there exists a bisimulation relation that relates x and y.

In lifting this notion of equivalence on hybrid transition systems to process terms (and hence abstracting from valuations) we have to be careful. It is assumed that the model variables that are shared by the process terms to be related represent the same entity. Therefore, both process terms are only compared with respect to the same (arbitrary) initial valuation of the model variables.

Definition 2.5 (Bisimilarity)
Two process terms $p, q \in \mathcal{T}(\mathcal{V}_{\mathrm{r}})$ are bisimilar, denoted $p \leftrightarroweq q$, if there exists a bisimulation relation $R \subseteq (\mathcal{T}(\mathcal{V}_{\mathrm{r}}) \times \mathit{Val}) \times (\mathcal{T}(\mathcal{V}_{\mathrm{r}}) \times \mathit{Val})$ such that $\langle p, \nu \rangle \,R\, \langle q, \nu \rangle$ for all valuations $\nu \in \mathit{Val}$.

As it turns out, this notion of equivalence leads to problems in combination with the parallel composition of processes. This is due to a possible sharing of variables between processes. As an

example, one might study the following discrete systems.

$$\begin{aligned} X &: \left[\ x \mid x^+ = 1\ \right] \gg a_1 \odot \left[\ x^- = 1\ \right] \gg a_2 \\ Y &: \left[\ x \mid x^+ = 1\ \right] \gg a_1 \odot a_2 \end{aligned}$$

Using bisimilarity, we find that $X \leftrightarroweq Y$, since the value of x is set to 1 by the first action a_1, the second re-initialization of X is irrelevant. However, placing these processes in parallel with the process

$$Z : \left[\ x \mid x^+ = 2\ \right] \gg a_3$$

clearly shows a difference. One might expect that $X \parallel Z \leftrightarroweq Y \parallel Z$, but the sharing of variables, leads to the result that the sequence a_1 followed by a_3 gives a deadlock situation for $X \parallel Z$, while the action a_2 can still occur for $Y \parallel Z$. The interference of Z shows a difference between X and Y.

In order for the equivalence to be robust with respect to interference caused by processes executed in parallel, for all states that are reached by performing transitions, it is required that the contained process terms are related for all valuations that can be obtained through interference. This is what we call robustness of a relation. An interference can be modeled as a function $\iota : \mathit{Val} \to \mathit{Val}$. Observe that we apply the same interference function to both variable valuations.

Definition 2.6 (Robust)
A relation $R \subseteq (\mathcal{T}(\mathcal{V}_\mathrm{r}) \times \mathit{Val}) \times (\mathcal{T}(\mathcal{V}_\mathrm{r}) \times \mathit{Val})$ is *robust* if for all $\langle p, \nu\rangle, \langle p', \nu'\rangle \in X$ such that $\langle p, \nu\rangle \, R \, \langle p', \nu'\rangle$, and for all interferences $\iota \in \mathit{Val} \to \mathit{Val}$, we find $\langle p, \iota(\nu)\rangle \, R \, \langle p', \iota(\nu')\rangle$.

Definition 2.7 (Robust bisimilarity)
Two process terms $p, q \in \mathcal{T}(\mathcal{V}_\mathrm{r})$ are robustly bisimilar, denoted $p \leftrightarroweq_r q$, if there exists a robust bisimulation relation $R \subseteq (\mathcal{T}(\mathcal{V}_\mathrm{r}) \times \mathit{Val}) \times (\mathcal{T}(\mathcal{V}_\mathrm{r}) \times \mathit{Val})$ such that $\langle p, \nu\rangle \, R \, \langle q, \nu\rangle$ for all valuations $\nu \in \mathit{Val}$.

Robust bisimilarity is selected as the core equivalence between hybrid processes.

2.2.2 Axiomatization

In this subsection, we give the axiomatization of robust bisimilarity in HyPA. In table 2.6, we give a set of derivation rules, and throughout this subsection we give a set of axioms that, to a large extend, capture the notion of robust bisimilarity. We write HyPA $\vdash_E p \approx_r q$, if we can derive equivalence of p and q using those axioms and recursive definitions from a set E.

Definition 2.8 (Derivation)
Let E be a set of recursive definitions over a set of recursion variables $\mathcal{V}_\mathrm{r}$. We write HyPA $\vdash_E p \approx_r q$ to indicate that equivalence of (open) terms p and q can be *derived* from our axiom system and the recursive definitions from E. In cases where there can be no confusion as to the set of recursive definitions that is intended, we write $\vdash$ instead of $\vdash_E$. We define that equivalence can be derived according to the rules given in table 2.6. In this table, p, p_i, q, q_i, r denote process terms, d, d' denote re-initialization clauses, and c, c', c'' denote flow clauses.

Rules (1), (2) and (3) of table 2.6 express that $\approx_r$ is an equivalence. Rules (4) and (5) express that it is a congruence. Rules (6) and (7) express that equivalence of flow predicates and re-initialization predicates transfers to equivalence of flow clauses and re-initialization clauses respectively. Rule (8) expresses that a recursive specification $X : p$ gives rise to an equivalence of X and p. In the remainder of this subsection, the axioms of HyPA, and the insight they provide regarding the operators of the language, are presented. Rule (9) expresses that these axioms

indeed define equivalences. In each of the axioms, x, y, z denote arbitrary terms. The letters a, a' denote actions, while c, c' denote flow clauses and d, d' denote re-initialization clauses. One may not choose δ when a is written in an axiom. At the end of this subsection, the intuitions behind derivation rules (10) and (11), are discussed.

Table 2.6: Derivation rules of HyPA

$$\frac{}{\text{HyPA} \vdash_E p \approx_r p}(1) \quad \frac{\text{HyPA} \vdash_E p \approx_r q}{\text{HyPA} \vdash_E q \approx_r p}(2)$$

$$\frac{\text{HyPA} \vdash_E p \approx_r q,\ \text{HyPA} \vdash_E q \approx_r r}{\text{HyPA} \vdash_E p \approx_r r}(3)$$

$$\frac{\text{HyPA} \vdash_E p \approx_r q,\ S : \mathcal{V}_\text{p} \to \mathcal{T}(\mathcal{V}_\text{p}),\ \text{dom}(S) \cap \mathcal{V}_\text{r} = \emptyset}{\text{HyPA} \vdash_E S(p) \approx_r S(q)}(4)$$

$$\frac{\mathcal{O} \text{ an n-ary HyPA operator},\ \forall_{1 \leq i \leq n}\ \text{HyPA} \vdash_E p_i \approx_r q_i}{\text{HyPA} \vdash_E \mathcal{O}(p_1, \ldots, p_n) \approx_r \mathcal{O}(q_1, \ldots, q_n)}(5)$$

$$\frac{\forall_{\nu,\nu'}\ (\nu, \nu') \models d \text{ iff } (\nu, \nu') \models d'}{\text{HyPA} \vdash_E d \gg x \approx_r d' \gg x}(6) \quad \frac{\forall_{\nu,\sigma}\ (\nu, \sigma) \models c \text{ iff } (\nu, \sigma) \models c'}{\text{HyPA} \vdash_E c \approx_r c'}(7)$$

$$\frac{X : p \in E}{\text{HyPA} \vdash_E X \approx_r p}(8) \quad \frac{p \approx_r q \text{ is an axiom}}{\text{HyPA} \vdash_E p \approx_r q}(9)$$

$$\frac{\begin{array}{c}\forall_{\nu,\sigma}\ (\nu, \sigma) \models c' \text{ implies } (\nu, \sigma) \models c \\ \forall_{\nu,\nu',\sigma}\ (\nu, \nu') \models d \text{ and } (\nu', \sigma) \models c \text{ imply } (\nu', \sigma) \models c'\end{array}}{\text{HyPA} \vdash_E d \gg c \approx_r d \gg c' \rhd c}(10)$$

$$\frac{\forall_{\nu,\sigma}\ (\nu, \sigma) \models c \text{ iff } (\nu, \sigma) \models c' \text{ or } (\nu, \sigma) \models c''}{\text{HyPA} \vdash_E c \approx_r (c' \oplus c'') \rhd c}(11)$$

The axioms of HyPA are presented in the tables 2.7 and 2.8. They were taken from [6] but rearranged and numbered for easy reference.

In each of the axioms, x, y, z denote arbitrary terms. The letters a, a' denote actions, while c, c' denote flow clauses and d, d' denote re-initialization clauses. One may not choose δ when a is written in an axiom.

2.2.3 Recursion principles

When reasoning about recursion, it is often useful to have a principle that claims that a solution of certain recursive specifications exists and is unique. That a solution exists follows directly from the operational semantics of HyPA, but it is not always clear that this particular solution is the only process satisfying the recursive equations. Let us first define what we mean by solution.

Definition 2.9 (Solution)
Let E be a recursive specification. An interpretation $S \in \mathcal{V}_\text{r} \to \mathcal{T}(\mathcal{V}_\text{r})$ of recursion variables as process terms, is a *solution* of E (denoted $S \models E$) if for every recursive definition $X : p \in E$ we have $S(X) \leftrightarrows_r S(p)$, where $S(p)$ denotes the process term induced by application of S to the variables of p. In particular, $S(X)$ is called a solution of $X : p \in E$.

The *recursive specification principle* RSP, which is quite standard in process algebra, states that so called guarded recursive specifications have at most one solution. For HyPA, guardedness of a recursive specification is defined as follows.

Definition 2.10 (Guardedness)
An open process term p is *guarded* if all occurrences of process variables in p, are in the scope of an action prefix $a \odot _$ or a flow prefix $c \rhd _$. A recursive specification E is *guarded* if for each recursive definition $X : p \in E$, p is derivably equal to a guarded process term, using the axiomatization of HyPA.

This leads to the principle given in table 2.9.

As an example, the process terms $\epsilon \oplus a \odot d \gg X$ and $c \rhd (X \oplus Y)$ are guarded, while the process terms $c \blacktriangleright X$ and $X \oplus a \odot X$ are not. That unguarded recursive equations do not necessarily have a unique solution, can be seen from the fact that the processes c and $(true)$ are both solutions of the equation $Y \approx_r c \blacktriangleright Y$. Also the equation $Z \approx_r Z \oplus a \odot Z$ has multiple solutions, some of which even execute flow transitions! Sadly, we do not have notation in HyPA to specify these unexpected solutions. From RSP, it follows that the set of equations $X_1 \approx_r \epsilon \oplus a \odot d \gg X_2$, $X_2 \approx_r c \rhd (X_1 \oplus X_2)$ has unique solutions for X_1 and X_2.

Moreover, it is proven that the solution of a guarded recursive specification is *bounded non-deterministic*.

Definition 2.11 (Bounded non-determinism)
Bounded non-determinism is recursively defined as:

- Every state has bounded non-determinism in 0 steps.
- A state (p, ν) has bounded non-determinism in $n+1$ steps, if for every l the set $R = \{(p', \nu') \mid \langle p, \nu \rangle \xrightarrow{l} \langle p', \nu' \rangle\}$ is finite, and all elements $(p', \nu') \in R$ have bounded non-determinism in n steps themselves.
- A state (p, ν) has bounded non-determinism (denoted $B(p, \nu)$) if it has bounded non-determinism for any arbitrary number of steps.
- A process p has bounded non-determinism (denoted $B(p)$) if for every valuation $\nu \in Val$ we find that (p, ν) has bounded non-determinism.

Van den Brand introduces a weaker definition of guardedness in [18]. To distinguish it from the previous definition of guardedness, we rename it to "subtitutably guarded":

Definition 2.12 (Substitutably guarded)
As in definition 2.10, an open process term p is *guarded* if all occurrences of process variables in p, are in the scope of an action prefix $a \odot _$ or a flow prefix $c \rhd _$. A recursive specification E is *substitutably guarded* if for each recursive definition $X : p \in E$, p is guarded, or can be transformed into a guarded term by replacing variables by the right-hand side of the equation.

The class of substitutably guarded specifications is contained in the class of guarded ones, which in turn is contained in the class of bounded non-derterministic specifications.

Table 2.7: Axioms of HyPA, part 1

1
$$\begin{array}{rcl} x \oplus y & \approx & y \oplus x \\ (x \oplus y) \oplus z & \approx & x \oplus (y \oplus z) \\ x \oplus x & \approx & x \\ (x \oplus y) \odot z & \approx & x \odot z \oplus y \odot z \\ (x \odot y) \odot z & \approx & x \odot (y \odot z) \end{array}$$

2
$$\begin{array}{rcl} x \oplus \delta & \approx & x \\ \delta \odot x & \approx & \delta \end{array} \qquad \begin{array}{rcl} \epsilon \odot x & \approx & x \\ x \odot \epsilon & \approx & x \end{array}$$

3
$$c \odot x \approx c \qquad [\mathit{false}] \approx \delta$$

4
$$x \blacktriangleright y \approx x \rhd y \oplus y$$

5
$$\begin{array}{rcl} (x \oplus y) \rhd z & \approx & x \rhd z \oplus y \rhd z \\ (x \rhd y) \rhd z & \approx & x \rhd (y \blacktriangleright z) \\ a \odot x \rhd y & \approx & a \odot (x \blacktriangleright y) \end{array} \qquad \begin{array}{rcl} \delta \rhd x & \approx & \delta \\ x \rhd \delta & \approx & x \\ \epsilon \rhd x & \approx & \epsilon \end{array}$$

6
$$x \parallel y \approx x \between y \oplus y \between x \oplus x \mid y$$

7
$$\begin{array}{rcl} \epsilon \between x & \approx & \delta \\ a \odot x \between y & \approx & a \odot (x \parallel y) \\ c \rhd x \between y & \approx & \delta \\ \delta \between x & \approx & \delta \\ \epsilon \mid \epsilon & \approx & \epsilon \\ \delta \mid x & \approx & \delta \\ \epsilon \mid c \rhd x & \approx & c \rhd x \\ \epsilon \mid a \odot x & \approx & \delta \end{array} \qquad \begin{array}{rcl} x \parallel \epsilon & \approx & x \\ (x \oplus y) \between z & \approx & x \between z \oplus y \between z \\ (x \between y) \between z & \approx & x \between (y \parallel z) \\ (x \mid y) \between z & \approx & x \mid (y \between z) \\ y \mid y & \approx & y \mid x \\ (x \oplus y) \mid z & \approx & x \mid z \oplus y \mid z \\ (x \mid y) \mid z & \approx & x \mid (y \mid z) \end{array}$$

$$\begin{array}{rcll} a \odot x \mid b \odot y & \approx & \gamma(a,b) \odot (x \parallel y) & \text{if } \gamma(a,b) \text{ defined} \\ a \odot x \mid b \odot y & \approx & \delta & \text{if } \gamma(a,b) \text{ undefined} \\ a \odot x \mid c \rhd y & \approx & \delta & \end{array}$$

$$c \rhd x \mid c' \rhd y \approx \left(\begin{array}{l} x \between c' \blacktriangleright y \oplus \\ y \between c \blacktriangleright x \oplus \\ x \mid c' \blacktriangleright y \oplus \\ y \mid c \blacktriangleright x \end{array} \right)$$

Table 2.8: Axioms of HyPA, part 2

8

$$\begin{array}{rcl} \partial_H(c) & \approx & c \\ \partial_H(\epsilon) & \approx & \epsilon \\ \partial_H(\delta) & \approx & \delta \\ \partial_H(a) & \approx & a \text{ if } a \in H \end{array} \qquad \begin{array}{rcl} \partial_H(x \oplus y) & \approx & \partial_H(x) \oplus \partial_H(y) \\ \partial_H(x \odot y) & \approx & \partial_H(x) \odot \partial_H(y) \\ \partial_H(x \rhd y) & \approx & \partial_H(x) \rhd \partial_H(y) \\ \partial_H(a) & \approx & \delta \text{ if } a \notin H \end{array}$$

9

$$\begin{array}{rcl} [true] \gg x & \approx & x \\ [false] \gg x & \approx & \delta \\ d \gg \delta & \approx & \delta \\ d \gg d' \gg x & \approx & (d \sim d') \gg x \\ d \gg x \oplus d' \gg x & \approx & (d \vee d') \gg x \\ c_{jmp} \gg c & \approx & c \end{array} \qquad \begin{array}{rcl} d \gg (x \oplus y) & \approx & d \gg x \oplus d \gg y \\ (d \gg a) \odot x & \approx & d \gg a \odot x \\ (d \gg c) \odot x & \approx & d \gg c \\ (d \gg \epsilon) \odot x & \approx & d^? \gg x \\ (d \gg x) \rhd y & \approx & d \gg x \rhd y \\ (d \gg x) \between y & \approx & d \gg x \between y \\ \partial_H(d \gg x) & \approx & d \gg \partial_H(x) \end{array}$$

10

$$\begin{array}{rcll} d \gg \epsilon \,|\, d' \gg \epsilon & \approx & (d^? \wedge d'^?) \gg \epsilon & \\ d \gg \epsilon \,|\, d' \gg a \odot x & \approx & \delta & \\ d \gg a \odot x \,|\, d' \gg a' \odot y & \approx & (d \wedge d') \gg \gamma(a,a') \odot (x \parallel y) & \text{if } \gamma(a,a') \text{ defined} \\ d \gg a \odot x \,|\, d' \gg a' \odot y & \approx & \delta & \text{if } \gamma(a,a') \text{ undefined} \\ d \gg \epsilon \,|\, d' \gg c \rhd x & \approx & (d^? \sim d') \gg c \rhd x & \\ d \gg c \rhd x \,|\, d' \gg a \odot y & \approx & \delta & \\ d \gg c \rhd x \,|\, d' \gg c' \rhd y & \approx & ((d \sim c_{jmp}) \wedge (d' \sim c'_{jmp})) \gg & \\ & & (c \wedge c') \rhd \begin{pmatrix} x \between c' \blacktriangleright y \oplus \\ y \between c \blacktriangleright x \oplus \\ x \,|\, c' \blacktriangleright y \oplus \\ y \,|\, c \blacktriangleright x \end{pmatrix} & \end{array}$$

Table 2.9: Recursive Specification Principle

$$\frac{S \models E,\ S' \models E,\ E \text{ guarded}}{S(X) \approx_r S'(X)}\ X \in \mathcal{V}_{\mathrm{r}}$$

Chapter 3

Requirements

This chapter lists all requirements for the HyPA simulator. First, the requirements are summarized in the next section. Then, all requirements are listed, including their motivation and current status. Finally, a table of requirements sorted by priority is given.

3.1 Overview

The simulator takes a HyPA model, and visualizes a trace of transitions that the model can perform. Since a HyPA model typically has many different possible traces, choices must be made at certain points in the simulation. The simulator is to have two modes of operation: step-wise simulation in which the user makes the choices, and random simulation in which the simulator makes the choices. The visualizations should give a clear impression of the continuous and discrete behavior of the model. The simulator should be easily extensible in some respects. For example, the architecture of the simulator should be such that other input file formats and other clause formalisms can be added easily.

3.2 Requirements

Each requirement is described by the following fields:

ID	A unique number for the requirement, for easy reference;
Priority	The importance of the requirement - a lower number means a higher importance;
Description	A description of the requirement;
Motivation	Why the requirement is present;
Status	The extent to which the requirement is fulfilled.

3.2.1 HyPA input

It is hard, if not impossible, to implement a simulator for all possible HyPA models. The requirements presented here indicate which classes of models should be handled correctly by the simulator.

First of all, there is the HyPA_{lin} class used by the linearization tool. Along the lines of e.g. pCRL, we also define a class of models called HyPA^{nt}. The idea behind this class is that the number of parallel processes stays the same throughout the simulation. The definitions of HyPA_{lin} models and HyPA^{nt} models can be found in appendix B.

ID: 1	**Priority:** 1	
Description: The input file format of the linearization tool from [18] can be read.		
Motivation: You do not want to rewrite a model just so that it can be input into two different tools. Therefore the simulator must be able to read models written for the existing tool.		
Status: Partially fulfilled. The file format that the simulator reads is the same as that of the linearization tool, with some extensions that could not be avoided (see section 5.2). Therefore, a file written for the linearization tool will have to be extended with some additional declarations before it can be simulated.		

ID: 16	**Priority:** 1	
Description: HyPA models in linear form (definition B.1) without variable abstraction can be simulated.		
Motivation: Linearized models are perceived as easy to simulate, since virtually no calculation of possible transitions needs to be performed. Because there already is a linearization tool, this would be a useful starting point for the simulator.		
Status: Fulfilled. Any HyPA specification can be simulated, and therefore linear ones as well. For the class of substitutably guarded models (definition 2.12), the simulator even finds *all* possible transitions. The currently implemented clause formalisms, however, do not allow for c_{jmp} constructs in the reinitialization clauses.		

ID: 43	**Priority:** 15	
Description: HyPA$^{\mathrm{nt}}$ models can be simulated.		
Motivation: Models of the form HyPA$^{\mathrm{nt}}$ frequently occur, and simulation of these models is anticipated to be simpler than simulation of guarded models in general (see section 4.1).		
Status: Fulfilled. Any HyPA specification can be simulated, and therefore the class HyPA$^{\mathrm{nt}}$ as well. For the class of substitutably guarded models (definition 2.12), the simulator even finds *all* possible transitions. The currently implemented clause formalisms, however, do not allow for c_{jmp} constructs in the reinitialization clauses.		

ID: 46	**Priority:** 16	
Description: HyPA$^{\mathrm{nt}}$ models with variable abstraction can be simulated.		
Motivation: It is useful that the simulator can read models with abstraction so that the abstractions need not be removed specifically for simulation. Variable abstraction does become less useful in simulation, as explained in section 4.2.4.2.		
Status: Partially fulfilled. For reasons also described in section 4.2.4.2, the simulator only supports one abstraction operator at the beginning of the model. This is enough to simulate the HyPA$_{\mathrm{lin}}$ models that can be read by the linearization tool. Therefore, models that are intended for analysis do not have to be rewritten for simulation.		

3.2.2 Clause formalisms

The HyPA specification does not say much about the form of flow and reinitialization predicates. We investigate what types of clauses occur in literature, and base requirements on that.

3.2.2.1 Clauses in literature

In this section we investigate what clauses occur in examples in literature. We use this knowledge to formulate requirements, and, in section 5.3, to design actual clause formalisms. The following example clauses were taken from [18], [6] and [14]. Some clauses were simplified.

- $\left(\ pumping \ \middle| \ true \ \right)$
- $\left(\ \neg(lastgatepos_{g2} = 0) \ \right)$
- $\left(\ gatepos_g \ \middle| \ (mode_g = \text{opening} \Rightarrow gatepos_g = V_{gate}) \wedge (mode_g = \text{closing} \Rightarrow gatepos_g = -V_{gate}) \ \right)$
- $\left[\ lastpressure_{c1}^{-} \leq P_{limit} \ \right]$
- $\left[\ pumping_c \ \middle| \ pumping_c^{+} = true \ \right]$
- $\left[\ lastpressure_c^{-} \leq P_{limit} \vee lastgatepos_{ga}^{-} \neq 0 \ \right]$
- $\left[\ Near(r^{-}, from) \ \right]$
- $\left(\ V \ \middle| \ V' = Q_i - Q_s \ \right)$
- $\left(\ Q_{min} \leq Q_s \leq Q_{max} \ \right)$
- $\left(\ Q_i = 0 \ \right)$
- $\left(\ t \ \middle| \ t' = 1 \wedge t \leq T \ \right)$
- $\left[\ V_{min} + V_{safe} \leq V^{-} \leq V_{max} - V_{safe} \ \right]$

Generalizing, we encounter the following constructs in clauses:

- General:
  - Model variables, constants, and literal values are strings, reals, or booleans;
  - Pre-defined functions and predicates are used;
- Reinitialization clauses:
  - Equalities ($=$);
  - Inequalities ($\neq < \leq \geq >$) ;
  - Boolean operators ($\neg \ \wedge \ \vee \ \Rightarrow$);
  - *true* and *false*;
- Flow clauses:
  - Equations ($Q_i(t) = 0$);
  - Ordinary differential equations ($V'(t) = Q_i(t) - Q_s(t)$);
  - Predicates on the value of a model variable ($\neg(lastgatepos_{g2}(t) = 0)$, $Q_s(t) \leq Q_{max}$);
  - *true* and *false*;
  - Boolean operators ($\neg \ \wedge \ \vee \ \Rightarrow$).

3.2.2.2 Requirements about clauses

We choose (differential) equations and inequalities for a clause formalism. Differential equations are a natural way to describe many physical processes. We anticipate the use of mathematical solvers such as Mathematica, Matlab or Maple to solve the equations. Different types of differential equations are described in appendix A. These definitions are used in the following requirements.

ID: 35	**Priority:** 1	
Description: The simulator can handle equations between the derivative of a model variable and other model variables and constants, e.g. $w' = v - s$, in flow clauses.		
Motivation: For a first version of the simulator, we like to keep the equations simple.		
Status: Fulfilled.		

ID: 36	**Priority:** 1	
Description: The simulator can handle equations between model variables and constants as flow predicates and reinitialization predicates.		
Motivation: For a first version of the simulator, we like to keep the equations simple.		
Status: Fulfilled.		

ID: 20	**Priority:** 10	
Description: The simulator can handle first-order linear differential equations in the flow clauses (definition A.7).		
Motivation: First-order linear differential equations are particularly easy to solve (the answer is of a fixed, known form) and already quite expressive.		
Status: Fulfilled.		

ID: 37	**Priority:** 11	
Description: The simulator can handle inequalities in flow clauses and reinitialization clauses.		
Motivation: Inequalities can be convenient, for example in the specification of the *clock* variable in the steam boiler model of section 7.1.		
Status: Fulfilled.		

ID: 39	**Priority:** 22	
Description: The simulator can handle all differential equations that the solver tool can handle.		
Motivation: It is useful to be able to use the knowledge that a certain mathematical solver is used.		
Status: Fulfilled.		

ID: 47	**Priority:** 25	
Description: The simulator can handle ordinary differential equations (definition A.3).		
Motivation: ODEs occur in existing example models in literature (see section 3.2.2.1).		
Status: Fulfilled		

3.2.3 Visualization

The purpose of the simulator is to visualize a trace through the state space of a given model. From [20] we already know some convenient visualizations. Motivation for the requirements is clear, so these requirements are given in condensed form.

ID:	Prio.:	Status:	Description:
6	3	Fulfilled	There are X/T plots of variables over time.
48	2	Fulfilled	Each X/T plot shows the names of the visualized variables in the caption.
49	2	Fulfilled	During a flow time choice, the X/T plots show clear indications of the minimum and maximum possible time length, if such a minimum or maximum exists.
21	1	Fulfilled	It is clear at which time the discrete actions take place.
50	2	Fulfilled	In a transition choice, all transitions are displayed. Double-clicking a transition chooses that transition and performs it.
52	7	Fulfilled	The jumps in model variable values are visualized when an action transition is chosen but not yet performed.
27	17	Fulfilled.	A legend can be shown in which the color for each model variable name is shown.
22	17	Fulfilled	The relation between two model variables can be visualized in an X/Y plot.
7	20	Not implemented	The current locations of the simulation of parallel processes is visible in the original model text.

Deleted requirements

The requirements below were deleted. Despite their initial priorities of 18 and 19, their importance was deemed so low that it became clear that they would never be implemented.

ID:	Prio.:	Status:	Description:
26	18	Deleted	The sequence of performed action transitions can be visualized in a message sequence chart. The user can indicate which action belongs to which process in the chart.
25	19	Deleted	The structure of the model can be visualized in a circle/square/line diagram analogous to diagrams of χ models.

3.2.4 Simulation

The requirements presented here define different modes of simulation. We have *step-by-step* simulation, in which the simulator stops whenever there is a choice to make. The user chooses a next transition and performs it, then chooses the next transition and so on. The other mode of simulation is *random*, in which the simulator makes each choice in a random way, until some stop

criterium is reached. Upon meeting the stop criterium, simulation resumes in step-by-step mode.

ID: 30	**Priority:** 1	
Description: Initial values for the model variables can be entered by the user.		
Motivation: The initial value for model variables is often left unspecified in models. In a simulation initial values must be chosen.		
Status: Fulfilled.		

ID: 8	**Priority:** 1	
Description: Step-by-step simulation is possible, i.e. every choice is left to the user.		
Motivation:		
Status: Fulfilled.		

ID: 9	**Priority:** 1	
Description: In step-by-step simulation, the visualizations of the model variables are up-to-date when a choice or a deadlock occurs.		
Motivation:		
Status: Fulfilled.		

ID: 34	**Priority:** 1	
Description: The choice for the time length of a flow clause can be made through slide bars in the X/T plots.		
Motivation: This method, designed by Huub van den Broek [20], has proven to be convenient.		
Status: Fulfilled.		

ID: 51	**Priority:** 1	
Description: The default choice for the time length of a flow clause is the maximum possible time length. This maximum time length is limited by a user-specified time limit.		
Motivation: See requirement 41 below.		
Status: Fulfilled.		

ID: 12	**Priority:** 4	
Description: Random simulation is possible, i.e. all choices are made by the simulator. The visualizations are updated continuously. The simulation is stopped on deadlock, termination and on user command.		
Motivation:		
Status: Fulfilled.		

ID: 44	**Priority:** 4	
Description: The stop criterium for random simulation can be the occurrence of a given action.		
Motivation:		
Status: Fulfilled. The user can select one or more actions that stop the random simulation.		

ID: 45	**Priority:** 12	
Description: The stop criterium for random simulation can be a predicate on the model variables.		
Motivation:		
Status: Not implemented yet.		

ID: 41	**Priority:** 13	
Description: The user can indicate that the maximum possible time interval is chosen for every flow transition in random simulation mode.		
Motivation: In many models, the most interesting trace is one in which all flows run for their maximum time length.		
Status: Fulfilled.		

ID: 13	**Priority:** 8	
Description: The user can undo steps whenever the simulation is stopped.		
Motivation:		
Status: Fulfilled.		

ID: 28	**Priority:** 9	
Description: The number of steps to undo can be indicated by the user in the X/T plots.		
Motivation:		
Status: Not implemented yet.		

ID: 31	**Priority:** 26	
Description: The simulator can be set up so that it fills in a random function for a given unconstrained model variable.		
Motivation: In a model on which analysis is performed, much of the world outside of a system is left unspecified. In the steam boiler example of section 7.1, for example, the flow of steam is specified as a flow clause ($s_{min} \leq s \leq s_{max}$). In simulation, you need to choose an actual behavior of the outside world. This requirement is meant to lighten the burden this places on the user.		
Status: Not implemented.		

ID: 32	**Priority:** 26	
Description: The simulator can be set up so that it fills in a random continuous function for a given unconstrained model variable.		
Motivation:		
Status: Not implemented.		

Deleted requirements

ID: 10	**Priority:** 4	
Description: Random simulation until a stop criterium is possible.		
Motivation:		
Status: Deleted - this requirement is already contained in the newer requirements 44 and 45.		

ID: 11	**Priority:** 4	
Description: In random simulation, the visualizations are updated at the end of the simulation.		
Motivation: In order to be able to simulate randomly as quick as possible, it may be convenient not to update the visualizations during the simulation.		
Status: Deleted. Updating the visualizations is much less time-consuming than the simulation process, and it serves as a progress indication.		

ID: 29	**Priority:** 18	
Description: The number of steps to undo can be indicated by the user in the message sequence charts.		
Motivation:		
Status: Deleted, Message Sequence Charts are no longer part of the requirements.		

3.2.5 Modularity

We anticipate several future extensions of the tool: other input file formats, additional types of visualizations, other equation solvers, and other clause formalisms.

ID: 2	**Priority:** 1	
Description: By adding a class and a reference, other file formats can be read.		
Motivation: Amending the simulator such that it reads other file formats should not be difficult. In order to make this requirement verifiable we specifically state that the addition of a class should be sufficient.		
Status: Partially fulfilled. New parsers can be added easily, but there is not yet a mechanism for choosing other file formats.		

ID: 4	**Priority:** 5	
Description: Another differential equation solver can easily be used.		
Motivation: Supporting only one mathematical solver severely limits the usability of the simulator, since not all users have access to the same solvers.		
Status: Not fulfilled. The only clause formalisms currently implemented are the 'mathematica' and the 'symbolicmma' clause formalisms (section 5.3). These formalisms use every possibility that Mathematica has to offer but they are hardly implementable on other solvers. Needed is a clause formalism in which a general syntax is defined, which is then translated to syntax that the different solvers understand. We once started working on such a formalism (section 5.3.1), but we abandoned it because of the inherent limitations on its expressiveness.		

ID: 23	**Priority:** 6	
Description: The system is extendable with a mode for numerical instead of symbolic computation.		
Motivation: Numerical solving of differential equations allows for a larger class of equations to be solved.		
Status: Fulfilled.		

ID: 19	**Priority:** 20	
Description: Another clause formalism can easily be added.		
Motivation: HyPA does not define the formalism used in the clauses. The usability of the solver would be prolonged if new clause formalisms could be added to it.		
Status: Fulfilled. Addition of a class and a factory method is all that is needed to add a new clause formalism.		

Deleted requirements

ID: 33	**Priority:** 1	
Description: The simulator can easily be changed to one that displays each plot in its own window.		
Motivation: Some simulators for other hybrid formalisms use separate windows.		
Status: Deleted - The prototypes and the first working version have shown that we do not want the X/T plots in separate windows. Apart from that, the requirement is not really verifiable.		

3.3 Requirements sorted by priority

In tables 3.1, 3.2 and 3.3, the requirements are listed by their priority. Deleted requirements are not in the table. The third column indicates the current status of the requirement: N (Not implemented), P (Partially implemented), or F (Fully implemented).

Priorities are assigned based on two criteria. Firstly, requirements should be independent of requirements with a lower priority. Secondly, the priorities are such that a system that meets the higher-priority requirements is relatively simple to build and yet usable. The combination of the requirements with priorities 1 through 3 yields a system that can perform step-wise simulation on linearized HyPA specifications that have relatively simple equations in the clauses. The requirements with priorities 4 through 9 specify features that make the simulator more convenient to use, with random simulation and undo-possibilities. The lower-priority requirements make the system more technically difficult, with more complicated clause formalisms and less restricted HyPA models.

Table 3.1: Requirements with the highest priorities, for a first working version

Prio	ID	St.	Description
1	1	P	The file format of the linearization tool from [18] can be read.
1	16	F	Linearized HyPA models can be simulated.
1	35	F	The simulator can handle equations between the derivative of a model variable and other model variables and constants.
1	36	F	The simulator can handle equations between model variables and constants as flow predicates and reinitialization predicates.
1	5	F	The value of model variables over time can be visualized (an X/T plot).
1	21	F	It is clear at which time the discrete actions have taken place.
1	30	F	Initial values for the model variables can be entered by the user.
1	8	F	Step-by-step simulation is possible, i.e. every choice is left to the user.
1	9	F	In step-by-step simulation, the visualizations of the model variables are up-to-date when a choice or a deadlock occurs.
1	34	F	The choice for the time length of a flow clause can be made through vertical slide bars in the X/T plots.
1	51	F	The default choice for the time length of a flow clause is the maximum possible time length.
1	2	P	By adding a class and a reference, other file formats can be read.
2	48	F	Each X/T plot should show the names of the visualized variables in the caption.
2	49	F	During a flow time choice, the X/T plots should show clear indications of the minimum and maximum possible time length.
2	50	F	In a transition choice, all transitions are displayed. Double-clicking a transition chooses that transition and performs it.

Table 3.2: Requirements with medium priorities

Prio	ID	St.	Description
4	12	F	Random simulation is possible, where all choices are made by the simulator. The visualizations are updated continuously. The simulation is stopped on deadlock and on user command.
4	44	F	The stop criterium can be the occurrence of a given action.
5	4	N	Another differential equation solver can easily be used.
6	23	F	The system is extendable with a mode for numerical instead of symbolic computation.
7	52	F	The jumps in model variable values are visualized when an action transition is chosen but not yet performed.
8	13	F	The user can undo steps whenever the simulation is stopped.
9	28	N	The number of steps to undo can be indicated by the user in the X/T plots.

Table 3.3: Requirements with low priorities

Prio	ID	St.	Description
10	20	F	The simulator can handle first-order linear differential equations in the flow clauses.
11	37	F	The simulator can handle inequalities in flow clauses and reinitialization clauses.
12	45	N	The stop criterium can be a predicate on the model variables.
13	41	F	The user can indicate that the maximum possible time interval is chosen for every flow transition in random simulation mode.
15	43	F	HyPA^{nt} models can be simulated.
16	46	P	HyPA^{nt} models with abstraction can be simulated.
17	22	F	The relation between two model variables can be visualized in an X/Y plot.
17	27	F	A legend can be shown in which each model variable name is replaced by a user-supplied name.
20	7	N	The current location in the original model text of each parallel process is visible.
20	19	F	Another clause formalism can easily be added.
22	39	F	The simulator can handle all differential equations that the solver tool can handle.
25	47	F	The simulator can handle ordinary differential equations.
26	31	N	The simulator can be set up so that it fills in a random function for a given unconstrained model variable.
26	32	N	The simulator can be set up so that it fills in a random continuous function for a given unconstrained model variable.

Chapter 4

Analysis

In this chapter, we explore some aspects of simulating a HyPA model. In section 4.1, we explore what HyPA models can be simulated, and how to obtain the transitions that a model can perform. Then, we introduce some new HyPA operators that make writing models easier and that increase the efficiency of the simulator.

4.1 First transitions

A HyPA model determines a transition system through its operational semantics. The simulator shows a trace through this transition system. Since the transition system can be infinitely large, we do not construct the entire transition system before picking a trace. Rather, we determine the outgoing transitions from the current state, pick one, and make the resulting state the current state. Although this set of outgoing transitions can be infinitely large as well, it turns out that we can describe this set in a finite way.

In this section, we provide a mechanism to determine the set of possible transitions that the HyPA model can perform, given the current state.

One solution is to use the HyPA linearization tool [18] to create a Linear Recursive Specification (definition B.1). From such a linear specification, the possible first transitions are more easily deductible than from a recursive specification in general. However, there are several drawbacks to this approach:

- Linearization is time-consuming;
- The resulting Linear Recursive Specification can be very large;
- There are severe restrictions on the models that the linearization tool can handle.

Another approach turns out to be simple and it allows for any substitutably guarded HyPA model (definition 2.12) to be simulated. From the HyPA operational semantics (section 2.1.1), an inductive function is derived that calculates the set of first transitions.

The transitions given by the operational semantics are tuples $\langle\, (p, v), (a, v'), (p', v'') \,\rangle$ or $\langle\, (p, v), \sigma, (p', v'') \,\rangle$, where:

- $(p, v), (p', v'')$ are states, with p, p' process terms and v, v'' valuations of the model variables;
- (a, v') is an action label consisting of an action a and a valuation v';
- σ is a flow.

Since the start state (p, v) is always the current state, we omit it from the results. In order to keep the set of transitions finite, we do not calculate the flows (σ) and end valuations (v''). We do provide the clauses that need to be solved to obtain the flows and valuations. This way, we allow the user to choose a transition first before solving the clauses.

4.1.1 First transitions function

Our function, named $F_E(t)$, calculates for a substitutably guarded (definition 2.12) HyPA specification $\langle\, t|E \,\rangle$ the set of transitions that can be performed, provided that their clauses can be solved. The transitions are denoted by tuples $\langle\, p, p' \,\rangle$. Process p indicates the clauses that need to be solved, and the type of transition. It has one of the forms:

$$d \gg a \quad | \quad d \gg c \quad | \quad d \gg \epsilon$$

Process p' is the process term of the resulting state (p', v'). In case of termination, the resulting state is denoted by $\checkmark$.

In the following definition, each line (except for the first one) can easily be traced back to a rule in the operational semantics.

Definition 4.1 ($F_E(t)$)
Let $a \in A,\ c \in C,\ d \in D,\ X \in \mathcal{V}_\mathrm{r},\ p, p', p'', q, q', q'' \in \mathcal{T}(\mathcal{V}_\mathrm{r})$, and $\langle t|E\rangle$ a substitutably guarded recursive specification. Note that p, p', p'', q, q', q'' cannot be equal to $\checkmark$ since this is not a process. Define the *first transitions of* $\langle t|E\rangle$, notation $F_E(t)$, inductively as follows:

$$
\begin{array}{lll}
(1) & F_E(\delta) & = \emptyset \\
(2) & F_E(\epsilon) & = \{\langle\, [true] \gg \epsilon, \checkmark \,\rangle\} \\
(3) & F_E(a) & = \{\langle\, [true] \gg a, \epsilon \,\rangle\} \\
(4) & F_E(c) & = \{\langle\, [true] \gg c, c \,\rangle\}
\end{array}
$$

$$
\begin{array}{lll}
(5) & F_E(d \gg p) & = \{\langle\, (d \sim e) \gg \epsilon, \checkmark \,\rangle \mid \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(p)\} \\
 & & \cup\ \{\langle\, (d \sim e) \gg p'', p' \,\rangle \mid \langle\, e \gg p'', p' \,\rangle \in F_E(p)\}
\end{array}
$$

$$
(6) \quad F_E(p \oplus q) = F_E(p) \cup F_E(q)
$$

$$
\begin{array}{lll}
(7) & F_E(p \odot q) & = \textbf{if } \exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)) \textbf{ then} \\
 & & \quad \{\langle\, d^? \wedge e^? \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, d^? \sim e \gg q'', q' \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, e \gg q'', q' \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, p'', p' \odot q \,\rangle \mid \langle\, p'', p' \,\rangle \in F_E(p)\} \\
 & & \textbf{else} \\
 & & \quad \{\langle\, p'', p' \odot q \,\rangle \mid \langle\, p'', p' \,\rangle \in F_E(p)\}
\end{array}
$$

$$
\begin{array}{lll}
(8) & F_E(p \blacktriangleright q) & = \{\langle\, d \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)\} \\
 & & \cup\ \{\langle\, p'', p' \blacktriangleright q \,\rangle \mid \langle\, p'', p' \,\rangle \in F_E(p)\} \\
 & & \cup\ F_E(q)
\end{array}
$$

$$
\begin{array}{lll}
(9) & F_E(p \rhd q) & = \{\langle\, d \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)\} \\
 & & \cup\ \{\langle\, p'', p' \blacktriangleright q \,\rangle \mid \langle\, p'', p' \,\rangle \in F_E(p)\}
\end{array}
$$

$$
\begin{array}{lll}
(10) & F_E(p \parallel q) & = \{\langle\, d^? \wedge e^? \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, (d_1 \sim c_{1jmp}) \wedge (d_2 \sim c_{2jmp}) \gg c_1 \wedge c_2, p' \parallel q' \,\rangle \\
 & & \qquad \mid \langle\, d_1 \gg c_1, p' \,\rangle \in F_E(p) \wedge \langle\, d_2 \gg c_2, q' \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, e^? \sim d \gg c, p' \,\rangle \mid \langle\, d \gg c, p' \,\rangle \in F_E(p) \wedge \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, e^? \sim d \gg c, q' \,\rangle \mid \langle\, d \gg c, q' \,\rangle \in F_E(q) \wedge \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(p)\} \\
 & & \cup\ \{\langle\, d \gg a, p' \parallel q \,\rangle \mid \langle\, d \gg a, p' \,\rangle \in F_E(p)\} \\
 & & \cup\ \{\langle\, d \gg a, p \parallel q' \,\rangle \mid \langle\, d \gg a, q' \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, d_1 \wedge d_2 \gg a, p' \parallel q' \,\rangle \\
 & & \qquad \mid \langle\, d_1 \gg a_1, p' \,\rangle \in F_E(p) \wedge \langle\, d_2 \gg a_2, q' \,\rangle \in F_E(q) \wedge \gamma(a_1, a_2) = a\}
\end{array}
$$

$$
\begin{array}{lll}
(11) & F_E(p \between q) & = \{\langle\, d \gg a, p' \parallel q \,\rangle \mid \langle\, d \gg a, p' \,\rangle \in F_E(p)\}
\end{array}
$$

$$
\begin{array}{lll}
(12) & F_E(p \mid q) & = \{\langle\, d^? \wedge e^? \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, (d_1 \sim c_{1jmp}) \wedge (d_2 \sim c_{2jmp}) \gg c_1 \wedge c_2, p' \parallel q' \,\rangle \\
 & & \qquad \mid \langle\, d_1 \gg c_1, p' \,\rangle \in F_E(p) \wedge \langle\, d_2 \gg c_2, q' \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, e^? \sim d \gg c, p' \,\rangle \mid \langle\, d \gg c, p' \,\rangle \in F_E(p) \wedge \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)\} \\
 & & \cup\ \{\langle\, e^? \sim d \gg c, q' \,\rangle \mid \langle\, d \gg c, q' \,\rangle \in F_E(q) \wedge \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(p)\} \\
 & & \cup\ \{\langle\, d_1 \wedge d_2 \gg a, p' \parallel q' \,\rangle \\
 & & \qquad \mid \langle\, d_1 \gg a_1, p' \,\rangle \in F_E(p) \wedge \langle\, d_2 \gg a_2, q' \,\rangle \in F_E(q) \wedge \gamma(a_1, a_2) = a\}
\end{array}
$$

$$
\begin{array}{lll}
(13) & F_E(\partial_H(p)) & = \{\langle\, d \gg a, \partial_H(p') \,\rangle \mid \langle\, d \gg a, p' \,\rangle \in F_E(p) \wedge a \notin H\} \\
 & & \cup\ \{\langle\, d \gg c, \partial_H(p') \,\rangle \mid \langle\, d \gg c, p' \,\rangle \in F_E(p)\} \\
 & & \cup\ \{\langle\, d \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)\}
\end{array}
$$

$$
(14) \quad F_E(X) = F_E(rhs(X))
$$

$F_E(t)$ has the following properties. The proofs for these theorems are given in appendix C.

Theorem 4.1 (Form of $F_E(t)$)
The value returned by $F_E(t)$ is a set of elements of the form

$$\langle\, d \gg a, p \,\rangle \quad | \quad \langle\, d \gg c, p \,\rangle \quad | \quad \langle\, d \gg \epsilon, \checkmark \,\rangle$$

Theorem 4.2 (Well-definedness)
$F_E(t)$ is well-defined for substitutably guarded HyPA models.

Theorem 4.3 (Soundness: termination)
Termination in $F_E(t)$ implies that $\langle\, t \,|\, E \,\rangle$ can terminate, i.e.:

$$\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark) \Rightarrow \langle\, t, v \,\rangle\checkmark$$

Theorem 4.4 (Soundness: transitions)
All transitions in $F_E(t)$ can be performed by $\langle\, t \,|\, E \,\rangle$, i.e.:

$$\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle) \Rightarrow \langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$$

Theorem 4.5 (Completeness: termination)
Termination of $\langle\, t \,|\, E \,\rangle$ is in $F_E(t)$, i.e.:

$$\langle\, t, v \,\rangle\checkmark \Rightarrow \exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$$

Theorem 4.6 (Completeness: transitions)
Every transition of $\langle\, t \,|\, E \,\rangle$, is in $F_E(t)$, i.e.:

$$\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle \Rightarrow \exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$$

4.1.2 Simulating other classes of HyPA models

Next to substitutably guarded models, we would like to simulate all guarded models (definition 2.10), and possibly even unguarded models. Soundness and well-definedness have to be maintained. Completeness is dropped, because infinitely many states may be reachable by a given transition in a not substitutably guarded model.

4.1.2.1 Guardedness

The definition of guardedness allows for models that can be rewritten into strictly guarded models by using axioms. The first transitions function is defined using the structure of the process term. Also, termination of the function is proven using the structure of the term. Therefore, we need some structural property that remains constant under axiomatic rewriting. This property should ensure that the function terminates.

Definition 4.2 (protected)
The occurrence of a recursion variable is *protected* iff it is in the scope of either $(t \odot _)$ or $(t' \between _)$ or $(t' \rhd _)$. Here, t must be such that not $\langle\, t,\ v \,\rangle\checkmark$ for any v, even if the clauses in t are satisfiable; t' can be any term.

The closest we have gotten to a property that remains constant is something that we call *protectedness* (definition 4.2). Note that strict guardedness (guardedness without axiomatic rewriting) implies protectedness. It appears to remain constant under all the current axioms, except for the ones concerning deadlock. For example, axiom 9-2 states that $\left[\ \mathit{false} \ \right] \gg x \approx \delta$. If we fill in a recursion variable for x, then the left process term is not protected while the right process term is. This is a problem, because it implies that the first transitions function does not terminate for all guarded process terms.

Another problem with this approach is that new axioms may be added, since the HyPA derivation system is still incomplete. If we would actually prove the invariance of a certain property under the axioms, this proof would become incomplete with each addition of a new axiom. Since any axiom has to satisfy robust bisimilarity, we could try to relate a structural property to a property of the underlying transition system. However, this approach has not been successful so far.

4.1.2.2 Fixpoint approach

A solution for termination of $F_E(t)$ could be to try and find a fixpoint. Suppose we have a mapping m of recursion variables to sets of first transitions. We replace rule (14) of $F_E(t)$ by $F_E(X) = m(X)$. Repeatedly we calculate $m(X) = m(X) \cup F(rhs(X))$ for each recursion variable $X \in \mathcal{V}_\mathrm{r}$ until m does not change. If we can prove that such a fixpoint exists and that it is reached, then we have a first transitions function that is well-defined.

This approach is not practical, as we describe next. We begin with some notational conventions from [4].

Partial order
We denote a partial order by $(D, \sqsubseteq)$. Here D is a set and $\sqsubseteq$ is a reflexive, anti-symmetric and transitive relation on $D \times D$. A partial order is *complete* if (1) it has a least element ($\bot$ such that $\forall_{d \in D} \bot \sqsubseteq d$), and (2) every chain (see below) has a least upper bound (see below).

Chain
Let $(D, \sqsubseteq)$ be a partial order. A *chain* with respect to $(D, \sqsubseteq)$ is an infinite sequence of elements $d_0, d_1, \ldots$ such that $d_0 \sqsubseteq d_1 \sqsubseteq \ldots$. We denote such a chain by $\langle d_i \rangle_{i \geq 0}$.

Least upper bound
An upper bound of a chain $\langle d_i \rangle_{i \geq 0}$ is an element d_n such that $\forall_{i \geq 0} d_i \sqsubseteq d_n$. For a *least* upper bound, we also have that for any other upper bound d_m, $d_n \sqsubseteq d_m$. When it exists, we denote the least upper bound by $\sqcup_{i \geq 0} d_i$.

Now we give some definitions of our own, and try to prove that the fixpoint approach works. This fails and we explain why.

Definition 4.3 (First transitions set)
We introduce a notation $\mathfrak{F}$ for the type of *first transitions*. A first transition is a pair $\langle \, p, \ p' \, \rangle$. Term p describes the transition and is of the form $d \gg \epsilon$ or $d \gg a$ or $d \gg c$. Term p' describes the resulting state after performing the transition. In case of termination, $p' = \checkmark$.

Definition 4.4 (First transitions mapping)
Define $\mathcal{M}$ as abbreviation for the type $\mathcal{V}_\mathrm{r} \to \mathcal{P}(\mathfrak{F})$, i.e. a mapping of recursion variables to sets of first transitions.

Definition 4.5 ($\sqsubseteq$)

Let $m, m' \in \mathcal{M}$. Now define the relation $\sqsubseteq$ on $\mathcal{M} \times \mathcal{M}$ as follows:

$$m \sqsubseteq m' \text{ iff } \forall_{X \in \mathcal{V}_\mathrm{r}}(m(X) \subseteq m'(X))$$

Lemma 4.1 ($(\mathcal{P}(\mathcal{M}), \sqsubseteq)$ is a CPO)
$(\mathcal{P}(\mathcal{M}), \sqsubseteq)$ is a complete partial order.

Proof
Trivially, $(\mathcal{P}(\mathcal{M}), \sqsubseteq)$ is reflexive, anti-symmetric, and transitive. Therefore it is a partial order. Furthermore, it has a least element $\overrightarrow{\emptyset} = \{X \to \emptyset \mid X \in \mathcal{V}_\mathrm{r}\}$. Finally, every chain $\langle s_i \rangle_{i \geq 0}$ has a least upper bound given by $\{X \to \bigcup_{i \geq 0} s_i(X) \mid X \in \mathcal{V}_\mathrm{r}\}$.

Definition 4.6 ($F_E^m(t)$)
Let m be a first transitions mapping. Redefine the first transitions function $F_E(t)$ to $F_E^m(t)$ by replacing rule (14) with

$$F_E^m(X) = m(X)$$

and replacing every occurrence of $F_E(p)$ by $F_E^m(p)$ for any p.

Definition 4.7 (First transitions of all recursion variables)
Let m be a first transitions mapping. Now define the function $\mathbb{F} : \mathcal{M} \to \mathcal{M}$ as follows:

$$\mathbb{F}(m) = \{X \to m(X) \cup F_E^m(rhs(X)) \mid X \in \mathcal{V}_\mathrm{r}\}$$

Now we would like to prove that $\mathbb{F}$ has a fixpoint and that that fixpoint is reached. This can be done by proving that $\mathbb{F}$ is continuous. Then, the theorem of Tarski/Knaster [4, 15] states that it has a unique least fixpoint given by $\sqcup_{i \geq 0} \mathbb{F}^i(\overrightarrow{\emptyset})$. For continuity, we need to prove that $\mathbb{F}$ is monotonic and that $\mathcal{P}(\mathcal{M})/_{\leftrightarrows_r}$ is finite. For finiteness of $\mathcal{P}(\mathcal{M})/_{\leftrightarrows_r}$ we need restrictions on the HyPA models.

Finiteness of the first transitions mapping reeks of bounded non-determinism (definition 2.11). For example, the not bounded non-deterministic process X, where $X : (X \oplus a) \odot X$ can reach an infinite number of states by performing one a-transition:

$$\begin{gathered} \epsilon \odot X \\ (\epsilon \odot X) \odot X \\ \dots \end{gathered}$$

Bounded non-determinism helps because it limits the possible number of states that can be reached by a transition. However, this does not limit the number of possible transitions from a given state.

In this respect we are helped by existing restrictions of the simulator. Firstly, the sets $\mathcal{A}$ and $\mathcal{V}_\mathrm{r}$ are finite, because the simulator has no syntax to specify infinite sets. Furthermore, the first transitions function abstracts from model variable valuations and flows. This implies that $\mathcal{P}(\mathcal{M})$ is finite with respect to robust bisimilation, for bounded non-deterministic processes.

However, in practice we cannot determine whether two processes are robustly bisimilar. For example, the bounded non-deterministic process $\partial_{\{a\}}(X)$, where $X : a \parallel X$, $\gamma(a,a) = c$, $\gamma(a,c) = \gamma(c,a) = a$, and $\gamma(c,c) = c$ can, with a c-transition, reach the syntactically different states

$$\begin{gathered} \epsilon \parallel X \\ \epsilon \parallel \epsilon \parallel X \\ \epsilon \parallel \epsilon \parallel \epsilon \parallel X \\ \dots \end{gathered}$$

All of these states are equivalent (axiom number 7-9). In practice, it prevents $\mathbb{F}$ from reaching a fixpoint and terminating.

This is not only a problem of the fixpoint approach, but also of the original first transitions function. The function returns a set, and thus it should not return two bisimilar transitions. In practice it does, but it terminates anyway. Some optimizations for this problem are described later in this chapter.

We abandon the fixpoint approach in favor of the *maximum recursion depth approach* described next.

4.1.2.3 Maximum recursion depth

The idea behind the maximum recursion depth approach is to allow any HyPA specification to be simulated, and let the user determine how much effort to spend on calculating first transitions. Well-definedness of the first-transitions function is trivial in this approach, completeness is only guaranteed for substitutably guarded models.

We add a parameter to $F_E(t)$ that indicates the number of times that F is applied to each recursion variable recursively. Also, we introduce a user-specified constant n_{max} that indicates the maximum allowed recursion depth.

Definition 4.8 ($F^n_E(t)$)
Let $n : \mathcal{V}_\mathrm{r} \to \mathbb{N}$. Redefine the first transitions function $F_E(t)$ to $F^n_E(t)$ by replacing rule (14) with:

$$(14) \quad F^n_E(X) \quad = \quad \textbf{if } n(X) = 0 \textbf{ then } \emptyset \textbf{ else } F^{n[n(X)-1/n(X)]}_E(rhs(X))$$

Here, $n[n(X) - 1/n(X)]$ means the mapping n with the value $n(X)$ replaced by $n(X) - 1$.

Definition 4.9 (n_{max})
n_{max} is a user-supplied constant mapping that maps all recursion variables to the same constant. This constant must be greater than 0.

$F^n_E(t)$ terminates: take weight function $w(n,t) = \langle\, n, o \,\rangle$ where o is the number of operators in t. This weight function decreases with respect to the lexicographical order, where $n_0 \leq n_1$ iff $\forall_{X \in \mathcal{V}_\mathrm{r}} n_0(X) \leq n_1(X)$.

The reason that we make n a mapping and not a single number, is to maintain completeness of $F^{n_{max}}_E(t)$ for substitutably guarded models. Because $n_{max}(X) \geq 1$, each recursion variable is allowed to be substituted for its right-hand side at least once.

A value of 1 for $n_{max}(X)$ is sufficient for substitutably guarded models. Suppose that $\langle\, t \,|\, E \,\rangle$ is substitutably guarded, and that a value of 1 is not sufficient. Because of $F_E(t)$-rules (7) and (9), $F^n_E(X)$ is only evaluated for unguarded occurrences of X. Since a value of 1 is assumed to

be insufficient, $F_E^n(X)$ needs to be called within a call to $F_E^n(X)$. This means that X occurs unguarded in its own equation. This implies that no finite number of substitutions of X by its right-hand side can make the equation guarded. Now we have a contradiction: the model is not substitutably guarded. This corresponds with the claim about $\rightarrow^u$ in [2, Note 2.3.9] and with the PNUDG principle of [16, Lemma 3.6.4].

4.1.3 Optimizations

The first transitions function as it is presented above turns out to take too much time to calculate. Also, solutions to c_{jmp} clauses are hard to calculate. There are several optimizations possible to make $F_E(t)$ faster and to prevent occurrences of c_{jmp}.

4.1.3.1 Eliminating c_{jmp}

The first transitions function introduces c_{jmp} constructs in rules 10 and 12 ($p \parallel q$ and $p \mid q$). It turns out that a c_{jmp} is difficult to solve. One needs to obtain all solutions to the flow clause. The Mathematica `NDSolve` function, however, requires that the initial values are known already to obtain only a single solution.

There is a frequently occurring case in which these jumps can be eliminated: if no variables are allowed to jump in the flow clauses, then $(d_1 \sim c_{1jmp}) \wedge (d_2 \sim c_{2jmp})$ equals $d_1 \wedge d_2$ [6, page 75, last paragraph]. Fortunately, any flow clause can be rewritten to a flow clause without jumps, as shown in theorem 4.7.

Theorem 4.7 (Eliminating jumps from flow clauses)
The flow clause

$$(\ V \ | \ pred \ )$$

is robustly bisimilar to

$$X, \text{ where } X : [\ \mathcal{V}_\mathrm{m} \backslash V \ | \ true \ ] \gg (\ \mathcal{V}_\mathrm{m} \ | \ pred \ ) \rhd X$$

Proof:
See appendix D.

The simulator does this transformation automatically. It does not solve c_{jmp}, so it requires that there is no actual occurrence of c_{jmp} in the HyPA model. This is not a very severe restriction, because the c_{jmp} operator was introduced for calculating with process terms rather than for specification.

4.1.3.2 Equality of resulting transitions

$F_E(t)$ returns a *set* of process term pairs. This raises the question of how to determine wether two process terms are equal. One problem lies in determining the equality of clauses. This would probably take as much time as solving them (if at all possible). An efficient approximation is simplifying the clauses using some simple rules, and then comparing them literally. Useful rules are:

$$\begin{aligned}
d \wedge [\ \mathcal{V}_\mathrm{m} \ | \ true \ ] &\rightarrow d \\
[\ V_1 \ | \ pred \ ] \wedge [\ V_2 \ | \ pred \ ] &\rightarrow [\ V_1 \cap V_2 \ | \ pred \ ] \\
d \sim [\ true \ ] = [\ true \ ] \sim d &\rightarrow d \\
d \vee d &\rightarrow d \\
d \vee [\ \mathcal{V}_\mathrm{m} \ | \ true \ ] &\rightarrow [\ \mathcal{V}_\mathrm{m} \ | \ true \ ] \\
[\ true \ ]^? &\rightarrow [\ true \ ]
\end{aligned}$$

These rules are chosen based on observations of the forms of clauses produced by $F_E(t)$. The rules are sound, i.e. the clauses on the left and right of each rule have the same solutions. This follows from the definition of a solution to a reinitialization clause (definition 2.3).

The comparison of process terms can be enhanced by taking commutativity and associativity into account. This still does not yield a complete equality function, but it already eliminates many duplicate transitions.

4.1.3.3 Dynamic programming

Evaluating an inductive function can lead to many evaluations of the same sub-term. A solution is to cache results.

In our case, $F_E^n(t)$ likely results in multiple evaluations of $F_E^n(X), X \in \mathcal{V}_\mathrm{r}$. It is good modeling practice not to write the same term twice in a model, but to use a recursion variable instead. Since the set E of recursion variable definitions does not change during simulation, it is very helpful to cache $F_E^n(X)$ for all $0 < n \leq n_{max}$ and $X \in \mathcal{V}_\mathrm{r}$.

4.2 Additional HyPA operators

Many real-life models have sub-processes that have a lot in common. In order to be able to specify such similar processes easily, we need some new operators.

The model variable abstraction operator defined in [18] is helpful but not enough. We also define an action renaming operator $\rho_A(p)$ and a model variable renaming operator $\varrho_V(p)$.

Besides ease-of-writing of HyPA models, the new operators allow for a more efficient implementation of the first-transitions function: when the first transitions for a process p are known, it is relatively easy to derive the first transitions for multiple renamed versions of p.

4.2.1 Action renaming

The action renaming operator renames actions in transitions. Its operational semantics are in table 4.1. The operator takes a map $A : \mathcal{A} \to \mathcal{A}$ and a process. Whenever the process can do an action-transition (a, ν), the renamed process can do the action transition $(A(a), \nu)$.

Table 4.1: Operational semantics of HyPA, action renaming

$$\frac{\langle\, p, \nu \,\rangle \checkmark}{\langle\, \rho_A(p), \nu \,\rangle \checkmark}(25) \qquad \frac{\langle\, p, \nu \,\rangle \overset{a,\nu'}{\to} \langle\, p', \nu'' \,\rangle}{\langle\, \rho_A(p), \nu \,\rangle \overset{A(a),\nu'}{\to} \langle\, \rho_A(p'), \nu'' \,\rangle}(26)$$

$$\frac{\langle\, p, \nu \,\rangle \overset{\sigma}{\rightsquigarrow} \langle\, p', \nu'' \,\rangle}{\langle\, \rho_A(p), \nu \,\rangle \overset{\sigma}{\rightsquigarrow} \langle\, \rho_A(p'), \nu'' \,\rangle}(27)$$

Theorem 4.8 (Robust bisimilation is a congruence for action renaming)
Robust bisimilation is a congruence for action renaming.

Proof
The OS-rules of HyPA, combined with the rules for action renaming, are in the *process-tyft* format of [11]. Therefore, stateless bisimilarity as defined in that article is a congruence for all HyPA

operators. As proven in [6, Appendix B], stateless bisimilation coincides with robust bisimilation and therefore robust bisimilation is a congruence as well.

4.2.2 Model variable renaming

The model variable renaming operator renames variables in transitions. Its operational semantics are in table 4.2. The operator takes a bijection $V : \mathcal{V}_\mathrm{m} \to \mathcal{V}_\mathrm{m}$ and a process. This must be a bijection, because all valuations in a model range over the same set $\mathcal{V}_\mathrm{m}$ of model variables. Therefore, you cannot rename two variables X and Y both to X, for example. Below, we use the following notational conventions:

- For a valuation ν, $V(\nu)$ indicates $\nu \circ V$;
- Likewise, for a flow σ, $V(\sigma)$ indicates $\sigma \circ V$;
- For a flow clause c, $V(c)$ is the clause c with all variables in its jump set and predicate replaced according to V;
- For a reinitialization clause d, $V(d)$ is the clause d with all variables replaced according to V;
- We use notation V^{-1} for the inverse of V. This inverse function exists because V in $\varrho_V(p)$ is required to be a bijective function.

Table 4.2: Operational semantics of HyPA, model variable renaming

$$\frac{\langle\, p, \nu \,\rangle \checkmark}{\langle\, \varrho_V(p), \nu \,\rangle \checkmark}(28) \qquad \frac{\langle\, p, \nu \,\rangle \overset{a,\nu'}{\to} \langle\, p', \nu'' \,\rangle}{\langle\, \varrho_V(p), V(\nu) \,\rangle \overset{a,V(\nu')}{\to} \langle\, \varrho_V(p'), V(\nu'') \,\rangle}(29)$$

$$\frac{\langle\, p, \nu \,\rangle \overset{\sigma}{\rightsquigarrow} \langle\, p', \nu'' \,\rangle}{\langle\, \varrho_V(p), V(\nu) \,\rangle \overset{V(\sigma)}{\rightsquigarrow} \langle\, \varrho_V(p'), V(\nu'') \,\rangle}(30)$$

Theorem 4.9 (Robust bisimulation is a congruence for model variable renaming)
Robust bisimulation is a congruence for model variable renaming.

Proof
The OS-rules for variable renaming are in the process-tyft format as well. Along the lines of theorem 4.8, we get that robust bisimulation is a congruence for model variable renaming as well.

4.2.3 First transitions function

To accommodate the new operators, we extend the definition of $F_E(t)$.

Definition 4.10 ($F_E(t)$, extension)
We extend the definition of the first transitions function with:

$$
\begin{array}{lrcl}
(15) & F_E(\rho_A(p)) & = & \{\langle\, d \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)\} \\
 & & \cup & \{\langle\, d \gg A(a), \rho_A(p'') \,\rangle \mid \langle\, d \gg a, p'' \,\rangle \in F_E(p)\} \\
 & & \cup & \{\langle\, d \gg c, \rho_A(p'') \,\rangle \mid \langle\, d \gg c, p'' \,\rangle \in F_E(p)\} \\
 & & & \\
(16) & F_E(\varrho_V(p)) & = & \{\langle\, V(d) \gg \epsilon, \checkmark \,\rangle \mid \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)\} \\
 & & \cup & \{\langle\, V(d) \gg a, \varrho_V(p'') \,\rangle \mid \langle\, d \gg a, p'' \,\rangle \in F_E(p)\} \\
 & & \cup & \{\langle\, V(d) \gg V(c), \varrho_V(p'') \,\rangle \mid \langle\, d \gg c, p'' \,\rangle \in F_E(p)\}
\end{array}
$$

In the proofs of the following theorems, it turns out that we need to assume that the solutions of a clause are not dependent on the name of the model variables, i.e. $(v, v') \models d$ iff $(V(v), V(v')) \models V(d)$, and $(v, \sigma) \models c$ iff $(V(v), V(\sigma)) \models V(c)$. This is a (very reasonable) restriction on the clause formalism.

Again, proofs of the theorems can be found in appendix C.

Theorem 4.10 (Form of $F_E(t)$)
The form of $F_E(t)$ as shown in theorem 4.1 is left intact by the new rules.

Theorem 4.11 (Well-definedness)
The extended version of $F_E(t)$ is well-defined.

Theorem 4.12 (Soundness: termination)
Termination in the extended version of $F_E(t)$ implies that $\langle\, t \mid E \,\rangle$ can terminate, i.e.:

$$\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark) \Rightarrow \langle\, t, v \,\rangle\checkmark$$

Theorem 4.13 (Soundness: transitions)
All transitions in the extended version of $F_E(t)$ can be performed by $\langle\, t \mid E \,\rangle$, i.e.:

$$\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle) \Rightarrow \langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$$

Theorem 4.14 (Completeness: termination)
Termination of $\langle\, t \mid E \,\rangle$ is in the extended version of $F_E(t)$, i.e.:

$$\langle\, t, v \,\rangle\checkmark \Rightarrow \exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$$

Theorem 4.15 (Completeness: transitions)
Every transition of $\langle\, t \mid E \,\rangle$, is in the extended version of $F_E(t)$, i.e.:

$$\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle \Rightarrow \exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$$

4.2.4 Model variable abstraction

Peter van den Brand has defined a model variable abstraction operator in [18, chapter 5]. First we give the original definitions. Then, we describe the extent of the support for the operator in our tool.

4.2.4.1 Definitions

For ease of reference, we repeat the definitions of model variable abstraction from [18, Chapter 5]. We re-number the OS-rules to distinguish them from the other HyPA OS-rules. The remainder of this subsection is from [18].

In table 4.3, the semantics of both types of abstraction operators is given. To keep the rules concise, the auxiliary functions $m_V(\mu, \nu)$ and $m_V(\sigma, \sigma')$ are used, which merge two valuations and two flows, respectively. The first function takes the valuations of the variables in the set V from ν and the valuations of all other variables from μ. Similarly, the second one takes the flows of the variables in the set V from σ' and the flows of all other variables from σ. Intuitively, this works like opening a new variable scope in a (procedural) programming language, because the newly introduced variables 'hide' existing variables with the same name. For any $\mu, \nu \in Val$, $\sigma, \sigma' \in \Sigma$ and $V \subseteq \mathcal{V}_\mathrm{m}$:

$$m_V(\mu, \nu)(n) = \begin{cases} \mu(n) & \text{if } n \notin V \\ \nu(n) & \text{if } n \in V \end{cases}$$

$$m_V(\sigma, \sigma')(n) = \begin{cases} \sigma(n) & \text{if } n \notin V \\ \sigma'(n) & \text{if } n \in V \end{cases}$$

Furthermore, $\nu \restriction_V$ denotes the valuation where $dom(\nu \restriction_V) = V$ and $\nu \restriction_V (n) = \nu(n)$ for all $n \in V$. $\overline{V}$ denotes the complement of V. Finally, $\sigma'(\uparrow)$ denotes the valuation of the flows in σ' in the last element of its domain.

Table 4.3: Operational semantics of HyPA, model variable abstraction

$$\frac{\langle\, p, m_V(\mu,\nu) \,\rangle \checkmark}{\langle\, |[\ V : \nu \mid p\ ]|, \mu \,\rangle \checkmark} \quad (31)$$

$$\frac{\langle\, p, m_V(\mu,\nu) \,\rangle \overset{a,w}{\mapsto} \langle\, p', w' \,\rangle}{\langle\, |[\ V : \nu \mid p\ ]|, \mu \,\rangle \overset{a, m_V(w,\mu)}{\mapsto} \langle\, |[\ V : w' \restriction_V \mid p'\ ]|, m_V(w', \mu) \,\rangle} \quad (32)$$

$$\frac{\langle\, p, m_V(\mu,\nu) \,\rangle \overset{\sigma}{\rightsquigarrow} \langle\, p', w' \,\rangle}{\langle\, |[\ V : \nu \mid p\ ]|, \mu \,\rangle \overset{m_V(\sigma,\sigma')}{\rightsquigarrow} \langle\, |[\ V : w' \restriction_V \mid p'\ ]|, m_V(w', \sigma'(\uparrow)) \,\rangle} \quad (33)$$

$$\frac{\langle\, |[\ V : \nu \mid p\ ]|, \mu \,\rangle \checkmark}{\langle\, |[\ V \mid p\ ]|, \mu \,\rangle \checkmark} \quad (34)$$

$$\frac{\langle\, |[\ V : \nu \mid p\ ]|, \mu \,\rangle \overset{a,w}{\mapsto} \langle\, p', w' \,\rangle}{\langle\, |[\ V \mid p\ ]|, \mu \,\rangle \overset{a,w}{\mapsto} \langle\, p', w' \,\rangle} \quad (35)$$

$$\frac{\langle\, |[\ V : \nu \mid p\ ]|, \mu \,\rangle \overset{\sigma}{\rightsquigarrow} \langle\, p', w' \,\rangle}{\langle\, |[\ V \mid p\ ]|, \mu \,\rangle \overset{\sigma}{\rightsquigarrow} \langle\, p', w' \,\rangle} \quad (36)$$

Rule (31) states that the abstraction of a terminating process can also terminate.

Rule (32) describes the case for a (discrete) action transition. This rule expresses an essential point of the abstraction operator, namely that the valuations of abstracted variables are not visible in the transition system. That is why the arrow in the conclusion is labeled with $m_V(w, \mu)$,

instead of simply w. Furthermore, the semantics is chosen such that the valuation of 'hidden' model variables does not change during an action transition. The reason for this choice is that in the existing semantics of HyPA, the valuations of model variables also do not change during an action transition.

Rule (33) describes the case for a (continuous) flow transition. This rule is similar to rule (2). Note however that the valuation in the resulting state is equal to the last valuation of the flow. Again, the reason for this choice is that in the existing semantics of HyPA, this is also the case.

Rules (34) to (36) define the actual abstraction operator, in terms of the auxiliary abstraction operator. Note that the local state variable ν in the hypotheses of these rules is an arbitrary valuation whose domain is V.

A (partial) axiomatization is given in Table 4.4. In these axioms, $Var(x)$ denotes the set of free variables in the term x. Table 4.4 only gives some basic axioms and some axioms that are necessary for the linearization algorithm.

Table 4.4: Axioms for the abstraction operator

$\|[\ V \mid x\ ]\| \oplus \|[\ V \mid y\ ]\|$	$\approx$	$\|[\ V \mid x \oplus y\ ]\|$	(VA1)
$\|[\ V \mid x\ ]\| \odot \|[\ V \mid y\ ]\|$	$\approx$	$\|[\ V \mid x \odot \left[\ V \;\middle|\; \mathit{true}\ \right] \gg y\ ]\|$	(VA2)
$\|[\ V \mid x\ ]\| \rhd \|[\ V \mid y\ ]\|$	$\approx$	$\|[\ V \mid x \rhd \left[\ V \;\middle|\; \mathit{true}\ \right] \gg y\ ]\|$	(VA3)
$\|[\ V \mid \partial_H(x)\ ]\|$	$\approx$	$\partial_H(\|[\ V \mid x\ ]\|)$	(VA4)
$\|[\ V \mid \|[\ W \mid x\ ]\|\ ]\|$	$\approx$	$\|[\ V \cup W \mid x\ ]\|$	(VA5)
$\|[\ V \mid x\ ]\|$	$\approx$	x, if $Var(x) \cap V = \emptyset$	(VA6)
$\|[\ V \mid x\ ]\| \parallel y$	$\approx$	$\|[\ V \mid x \parallel y\ ]\|$, if $Var(y) \cap V = \emptyset$	(VA7)
$d \gg \|[\ V \mid x\ ]\|$	$\approx$	$\|[\ V \mid d \gg x\ ]\|$, if $Var(d) \cap V = \emptyset$	(VA8)
$\|[\ v \mid x\ ]\|$	$\approx$	$\|[\ w \mid x[{}^{w}/_{v}]\ ]\|$, if $w \notin Var(x)$	(VA9)

Axioms (VA1), (VA2), (VA3) and (VA4) all express distribution of the abstraction operator over other operators. Together, they describe how abstraction can be distributed over (closed) linear terms.

Axiom (VA5) describes how two abstractions can be merged. Axiom (VA6) states that abstracting from variables that do not occur freely in the abstracted term has no influence. Both of these axioms are useful for introducing abstractions or for eliminating them.

Axioms (VA7) and (VA8) expresses that a parallel term or re-initialization clause can be pulled into the abstraction, as long as the abstracted variables do not occur freely in that term or clause. These axioms appeal to the same intuition as axiom (VA6). However, they cannot simply be derived from (VA6), because abstraction does not distribute over parallel composition or re-initialization in general.

Axiom (VA9) states that abstracted variables can be renamed (α-conversion). Note that in this axiom v and w denote single variables, not sets of variables. The expression $x[{}^{w}/_{v}]$ denotes the substitution of w for all free occurrences of v in the process term x.

4.2.4.2 Simulator support for abstraction

As noted in the section about model variable renaming, all valuations have domain $\mathcal{V}_{\mathrm{m}}$. Therefore, there must be a "global" variable for each abstracted variable. Thus, the abstraction operator abstracts from variable behavior, not from the variables themselves.

We would like to be able to simulate all models that can be linearized, i.e. $\mathrm{HyPA}_{\mathrm{lin}}$ models (appendix B). These models can have one occurrence of an abstraction operator, namely at the

top level. This means that the behavior of some variables is hidden. During simulation, the behavior of all variables (global and abstracted) needs to be specified. This means that the simulator must bother the user about initial values and behavior for abstracted variables anyway. To avoid bothering the user about the redundant global variables as well, we can implement abstraction for HyPA_{lin} models simply by erasing it from the model.

We do not implement full support for the abstraction operator for a number of reasons:

1. The user needs to give initial values for all variables, including abstract ones. This means that for simulation, abstraction does not really work;
2. Variables can already be removed from the graphical representations of the behavior;
3. It is unlikely that the abstraction operator will be used in models, since the linearization tool does not support it fully;
4. Next to the new renaming operators, abstraction is not needed to simplify writing of models;
5. Considering the drawbacks and usability, it is a lot of work to implement abstraction.

Chapter 5

Design

This chapter describes the higher-level design decisions made in building the simulator. This includes the architecture of the system, the input file format, and clause formalisms.

5.1 Architecture

The architectural design of the simulator is described here. We use the UML static structure symbols depicted in figure 5.1. Our terminology includes the design patterns defined in [8]. We give a short explanation of the used design patterns:

Façade One single class provides the interface to a complex program part. This class itself provides no functionality, but is there to simplify the use of the program part.

Factory This pattern is used to separate the creation of subclass instances from the knowledge of which subclass to use. This knowledge is within a separate Factory class. An object is created by invoking a Factory class method. Factory subclasses implement the creation and return objects of different subclasses.

Observer Used for notifying multiple "observers" of changes in a portion of the state (the "subject"). Each observer registers itself with the subject, and is thereafter notified of any change in the subject.

Singleton Used when there may be only one instance of a class. It involves hiding the constructor of the class, and access to the instance is provided through a static method. Access to the instance is hereby given to every part of the program that knows about the existence of the class.

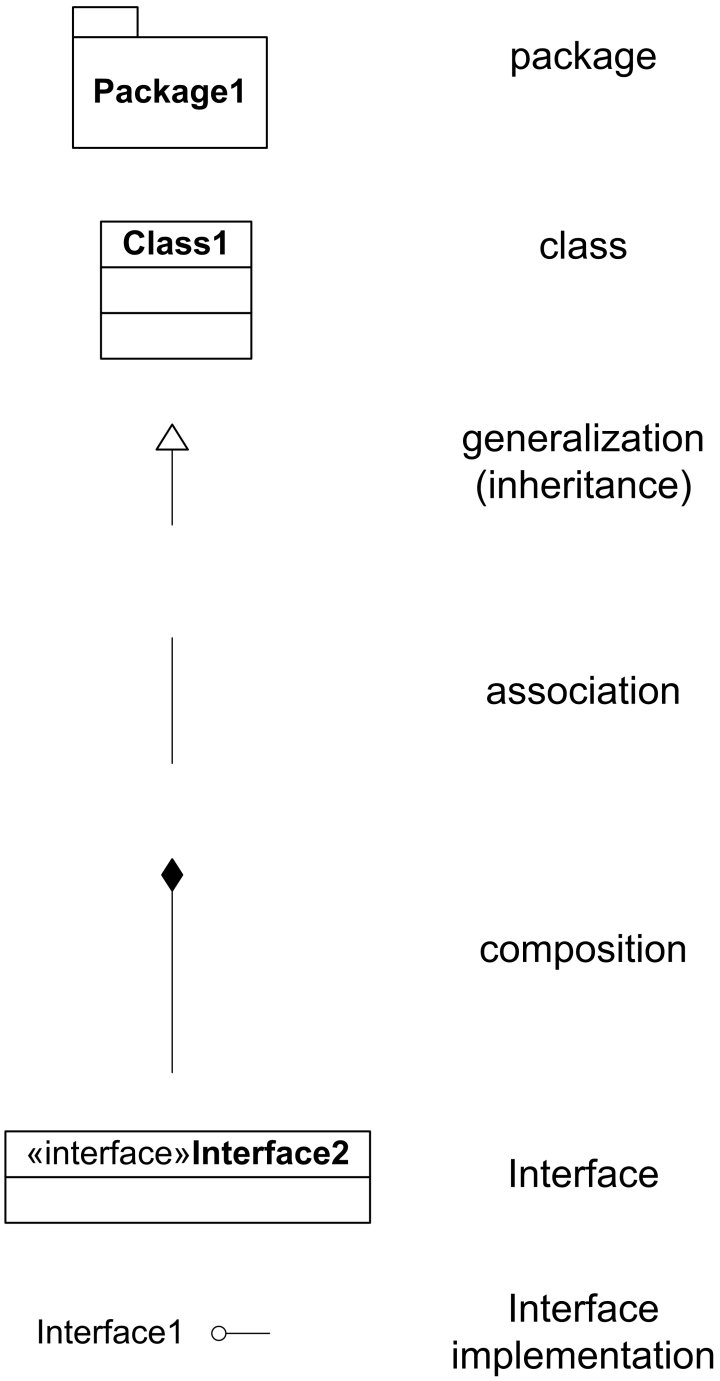
Visitor A pattern used to traverse trees of objects. Each tree node guides the visitor class to its sub-nodes. The visitor has a method for each type of node that it can encounter. Upon reaching a node, the corresponding method is invoked.

First we make a rough division of the system into parts. Then we describe the design of each part.

5.1.1 Rough division

The user interface of the simulator is built first, in the eXtreme Programming fashion [3]. This way, we know that we build what the user wants, and we find out exactly what is needed underneath the user interface. This calls for an architectural division of the software into two parts "ui" and "engine".

Figure 5.1: Legend

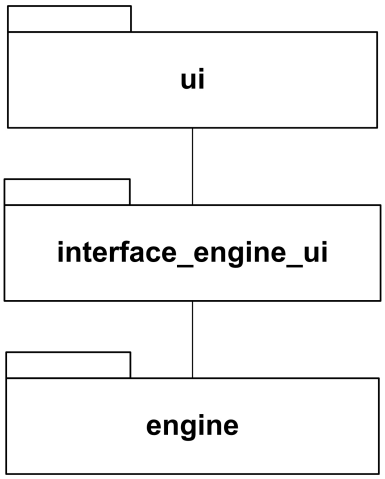


Design decision 5.1 (Rough division)
The system is divided into two parts. The "user interface" part contains all dialogs, buttons and visualizations. It is responsible for coordinating all visualizations and for receiving user input. The "engine" performs all calculations: parsing, determining first transitions, and solving clauses.

This division is visible in the package structure of the program (figure 5.2).

Design decision 5.2 (Package structure)
We have three top-level packages: `ui` for the "ui" part, `engine` for the "engine" part, and an `interface_engine_ui` package in between. The latter package contains all classes for communicating between the former two. As a general rule, classes in the `ui` package only know about the top two packages, and classes in the `engine` package only know about the bottom two packages. This is useful as a protection mechanism: the user interface classes cannot reference anything inside the engine and vice versa, so the division is enforced by the compiler.

Figure 5.2: Rough package division



5.1.2 User interface

The user interface part has the following tasks:

- Receiving user input;
- Enabling and disabling the controls for user input;
- Visualization of the current state;
- Synchronizing the position, zooming and scrolling of the visualizations;
- Initiating different modes of simulation: manual or random.

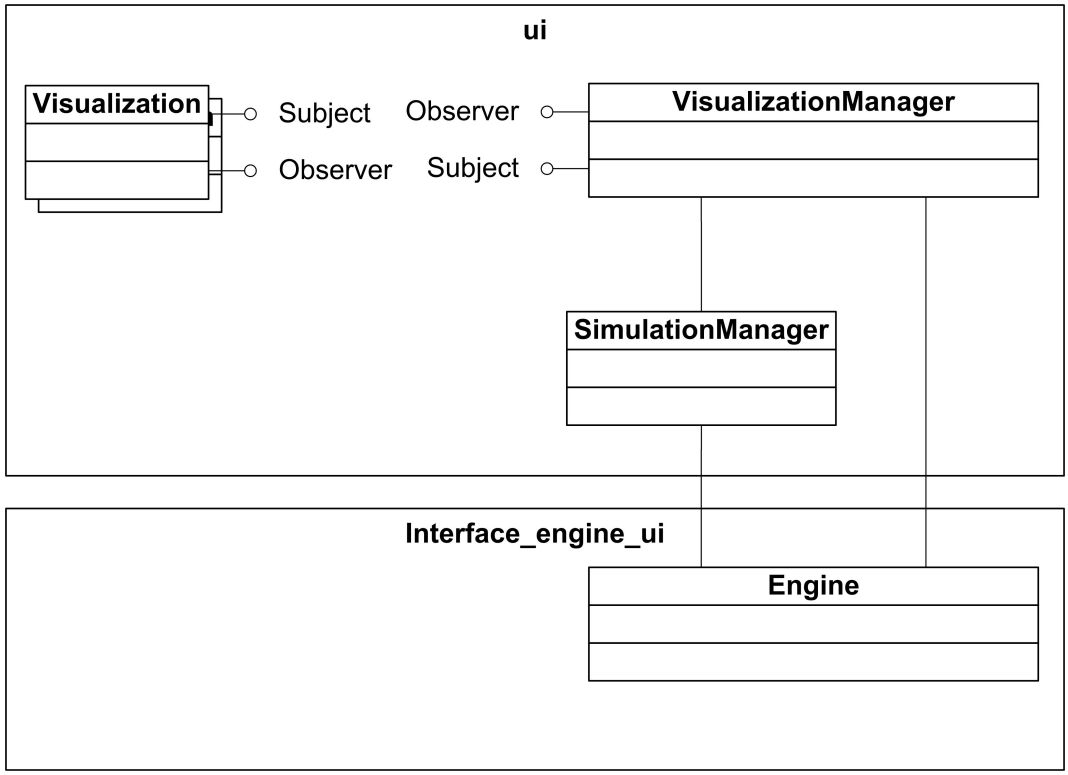
Design decision 5.3 (UI division)
We divide the user interface into (figure 5.3):

1. Visible controls for input and visualization; these include forms, buttons, plots etc.;

2. A `VisualizationManager` class that coordinates the visible controls;

3. A `SimulationManager` class that controls the mode of simulation. It controls when first transitions are calculated, which of them are solved, and it has a thread for random simulation.

Figure 5.3: User interface static structure overview



5.1.2.1 Visualizations

The different visual controls, such as X/T plots, X/Y plots, the legend, the transition choice box etc, require different sets of information about the current state and state changes.

Design decision 5.4 (Use of observer pattern)
In order to keep the design extensible and simple, we use the Observer pattern [8] to couple the controls to the visualization manager. The controls, depending on what information they need, implement one or more of the interfaces in table 5.1. Each control is registered with the visualization manager for the information that it needs. After registering, it is notified of new information.

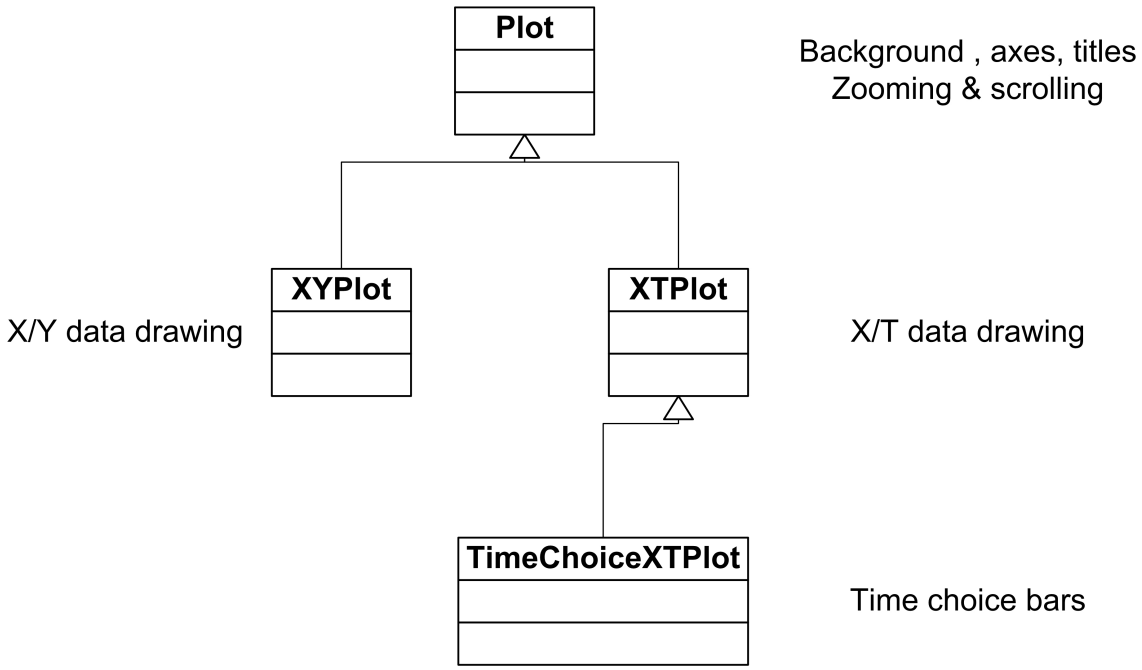
In the prototypes, we used a freely available library to create plots. However, extending these plots with time choice bars proved difficult, so we implemented our own plots. The class hierarchy

is in figure 5.4. A basic `Plot` class contains all code common to X/T and X/Y plots. Then, the classes `XTPlot` and `XYPlot` implement the drawing of data on the plot. The `TimeChoiceXTPlot` class adds time choice bars. The `XTPlot` class is never used in practice, because we want time choice bars on all X/T plots. The reason for this extra class is simply to make the code more readable: the functionality is divided over more classes.

Table 5.1: Observer interfaces

ISpecificationOberver	Loading and closing of specifications;
IEnableObserver	Enabling and disabling of possible user actions;
ITransitionChoiceObserver	Choice of a transition;
ITimeChoiceObserver	Choice of a time length of a flow transition;
IPerformObserver	Performing of transitions;
IXAreaObserver	Changing the position and size of X/T plots;
IZoomScrollObserver	Zooming and scrolling in X/T plots.

Figure 5.4: Plot class hierarchy



5.1.2.2 User input

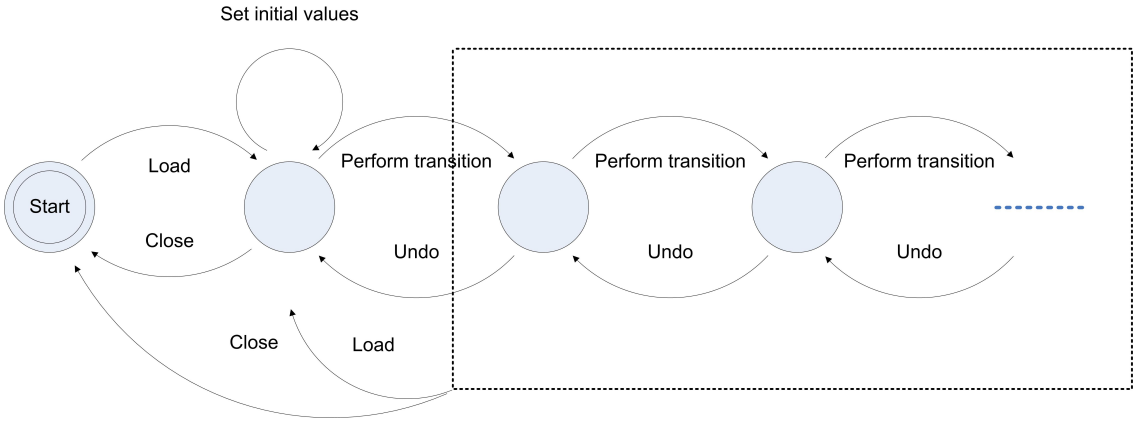
The `VisualizationManager` translates all user commands into corresponding commands for the engine or simulation manager, and notifies the visualizations of any changes in the current state. When necessary, it pops up dialogs for the user to enter extra information.

Design decision 5.5
For each user command (load specification, perform this transition etc.) there is a method in

`VisualizationManager`. This keeps the visible controls and forms light-weight: all they need to do is call one method for each action that the user performs.

Typical user actions are shown in figure 5.5. The user loads a specification, sets initial values, and starts performing and undoing transitions. Of course, only performed transitions can be undone. Finally, the current specification is closed and/or a new one is loaded.

Figure 5.5: User command statechart



5.1.2.3 Simulation

The `SimulationManager` class controls the mode of simulation. The visualization manager directs most user commands directly to the engine, but not the retrieval of the set of first transitions. Instead, the simulation manager retrieves the first transitions, solves them, and eliminates transitions without solutions. An example of the communications involved is in figure 5.6.

Random simulation is achieved through the creation of a thread (figure 5.7). The thread communicates with the engine. It repeatedly retrieves the set of first transitions, picks one, and solves and performs it. Every so many seconds, it communicates all state changes back to the visualization manager.

Having an extra thread implies that some form of synchronization is needed. The simulation manager communicates with the engine and with the visualization manager.

Design decision 5.6 (Synchronization)
The engine has all its public methods declared `synchronized` so that no two threads can access it at the same time. The communication to the visualization manager is buffered and synchronized so that only the main thread manipulates visual controls.

5.1.3 Interface

We define the interface between user interface and engine in this section. This interface consists of an abstract class for accessing the engine, representations for transitions, process terms, solutions,

Figure 5.6: First transitions communications

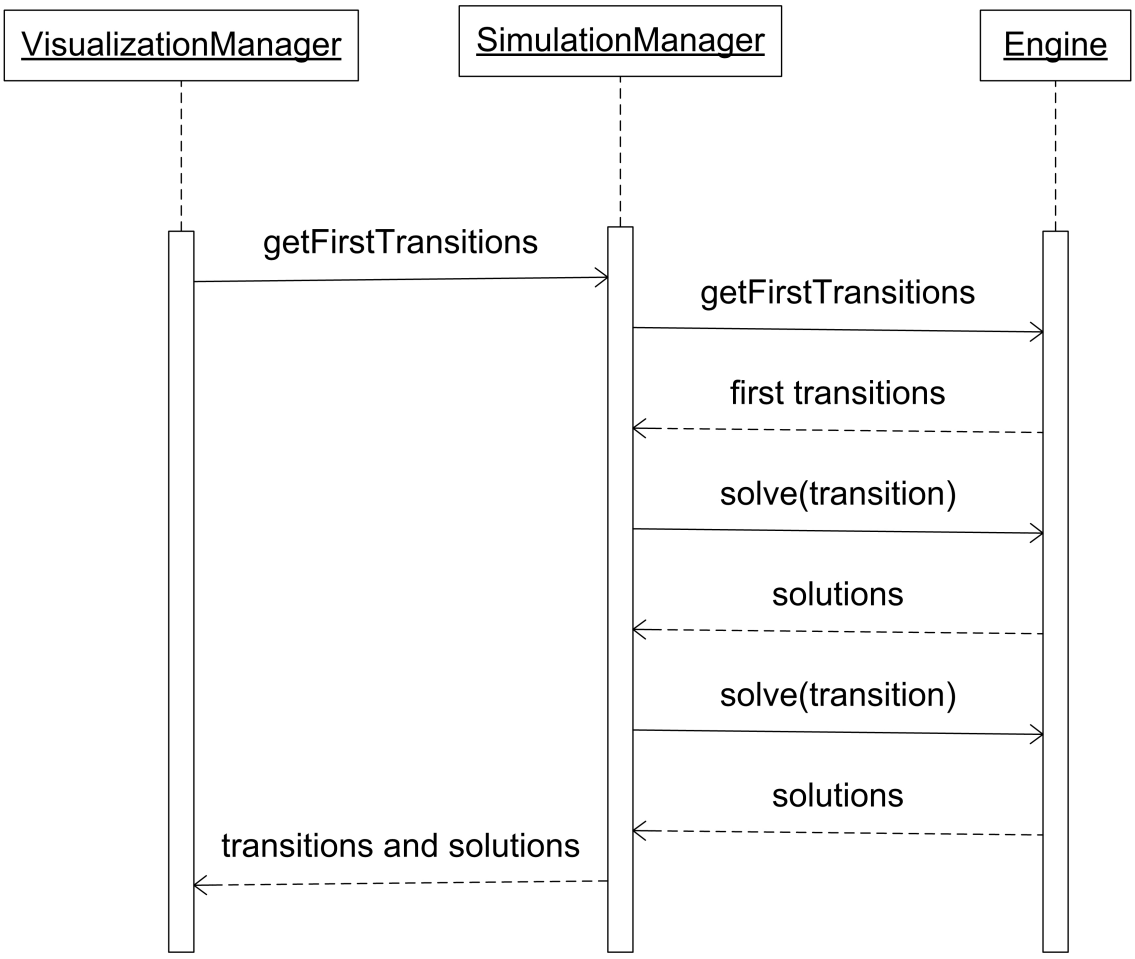
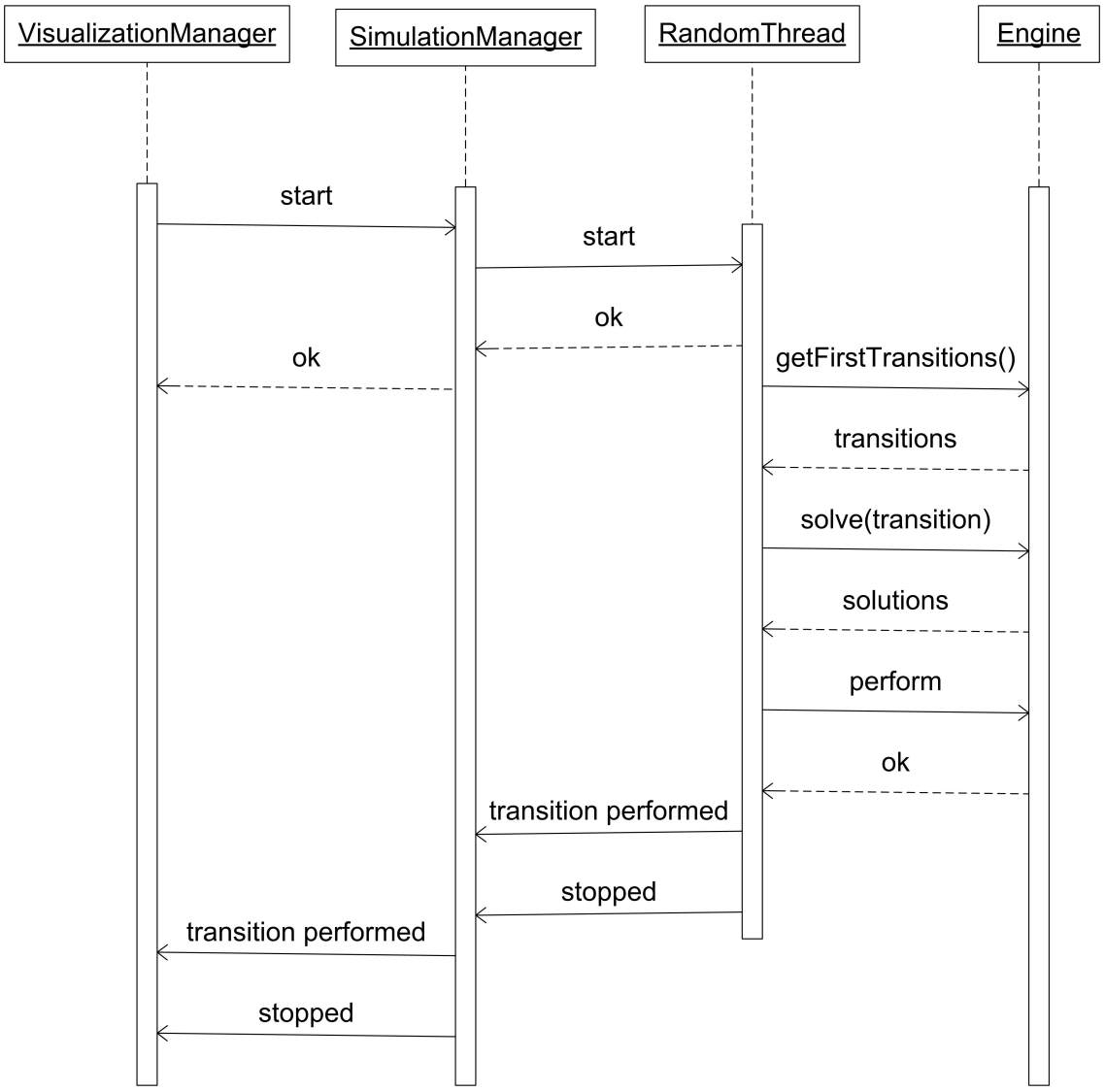


Figure 5.7: Random simulation communications



times and values, and exceptions. Each is described below.

5.1.3.1 Engine façade

In order to be able to test the user interface before implementing the rest, we define an interface through which all communications to the "engine" take place. This single access point to the possibly complicated engine is also known as a Façade [8]. It creates the possibility of attaching a dummy engine or a real engine depending on what we want to test. Also, it hides the complexity of the actual engine implementation.

Design decision 5.7 (Abstract `Engine` class)
There is a Façade to the engine that consists of an abstract class named `Engine`. Subclasses `DummyEngine` and `RealEngine` then implement the different engines (figure 5.8). The class has the methods listed in table 5.2.

More detailed specifications of the methods can be found in the implementation documentation.

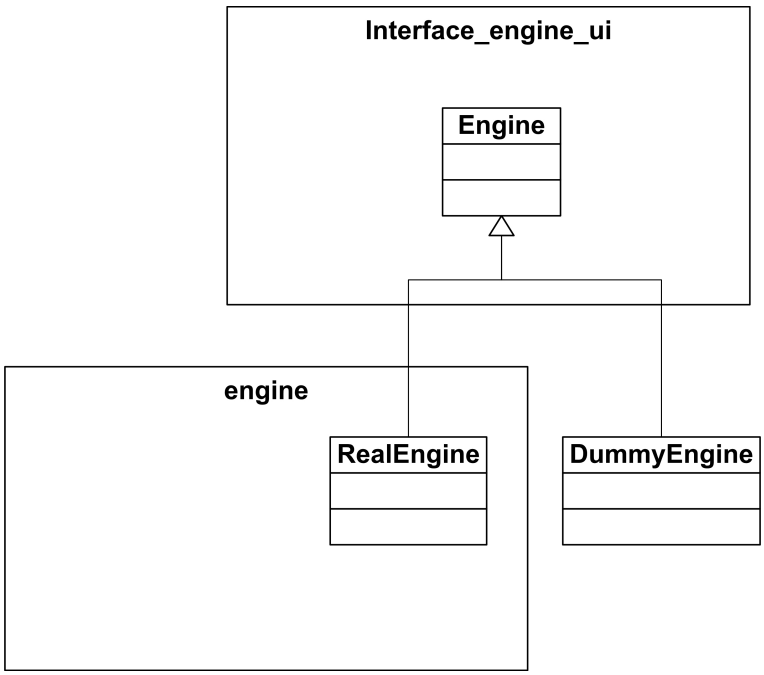
There are restrictions to the order in which the methods may be called. These correspond to the user command restrictions of figure 5.5. First, a specification must be loaded. Then, all other methods may be called. Initial values may only be set before performing any transitions. Transitions may only be undone after performing them.

Design decision 5.8 (Singleton `Engine` class)
In order to make the current `Engine` instance available to the many classes of the `ui` package, we use the Singleton design pattern [8]. The abstract class `Engine` has an `initialize` method that takes a subclass instance. This method is to be called once. After that, a public static method `instance` provides access to the sole instance.

Table 5.2: Engine methods

loadSpecification(filename)	Loads a HyPA specification from a given file.
getSpecificationText()	Returns the original specification text.
getSpecification()	Returns the specification as an object.
getModelVariables()	Returns a list of all model variables.
getCurrentTime()	Returns the sum of all performed flow transition lengths.
getInitialValues()	Returns the initial value for each model variable.
setInitialValues(values)	Sets new initial values for the model variables.
getFirstTransitions()	Retrieves the transitions that the model can perform next.
solve(transition)	Solves the clauses of a given transition.
performTransition(transition, solution)	Performs a given transition.
undo(n)	Undoes the last n performed transitions.
closeSpecification()	Closes the current specification.

Figure 5.8: Engine façade



5.1.3.2 Transitions

The `getFirstTransitions` method of the `Engine` returns a set of `Transition` objects. As described in section 4.1, a transition returned by $F_E(t)$ is a tuple $\langle p, p' \rangle$. Process p indicates the clauses that need to be solved, and the type of transition. It has one of the forms:

$$d \gg a \quad | \quad d \gg c \quad | \quad d \gg \epsilon$$

Process p', called *next process*, is the process term of the resulting state (p', v'). In case of termination, the resulting state is denoted by $\checkmark$.

In our application, the `Transition` object is more than just that tuple. Firstly, we frequently have sets of multiple tuples $\langle p, p'_1 \rangle, \ldots, \langle p, p'_n \rangle$ with the same process p. As an optimization, we represent these tuples by a single `Transition` object that contains process p with a set of processes $\{p'_1, \ldots, p'_n\}$.

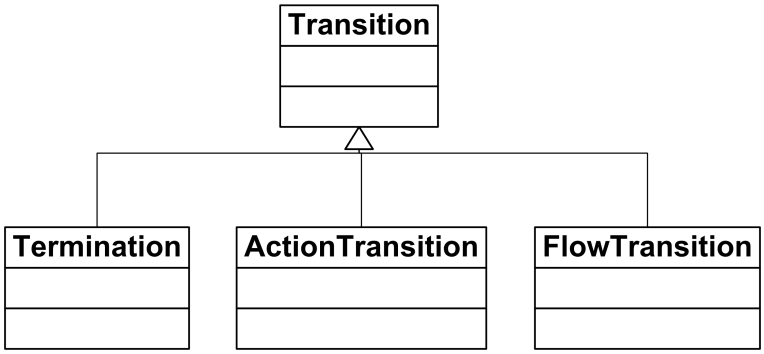
Secondly, a place is needed to store solutions to the transitions as well. The user interface retrieves the possible transitions from the engine, and may request the engine to solve some or all of them. After that, the user must select the combination of a *transition*, *solution* and a *next process* to perform.

Thirdly, sometimes additional predicates are needed. In many models intended for analysis, the clauses do not completely specify the behavior of the model variables. An example is the flow clause predicate $s_{min} \leq s \leq s_{max}$. In this case the user must enter some additional predicates to constrain the behavior of s for simulation. These predicates are also stored in the `Transition` object.

Design decision 5.9 (`Transition` class)
The `Transition` class represents a tuple $\langle p, A, N, S \rangle$, where p indicates the clauses and type of transition ($d \gg a \quad | \quad d \gg c \quad | \quad d \gg \epsilon$), A is a set of additional predicates, N is a set of possible next process terms, and S is a set of solutions to the clauses.

Design decision 5.10 (Transition hierarchy)
There are some differences between the three types of transitions, and therefore a subclass is created for each of them. The hierarchy can be found in figure 5.9.

Figure 5.9: Transition hierarchy



5.1.3.3 Process terms

Throughout the application, HyPA process terms must be represented.

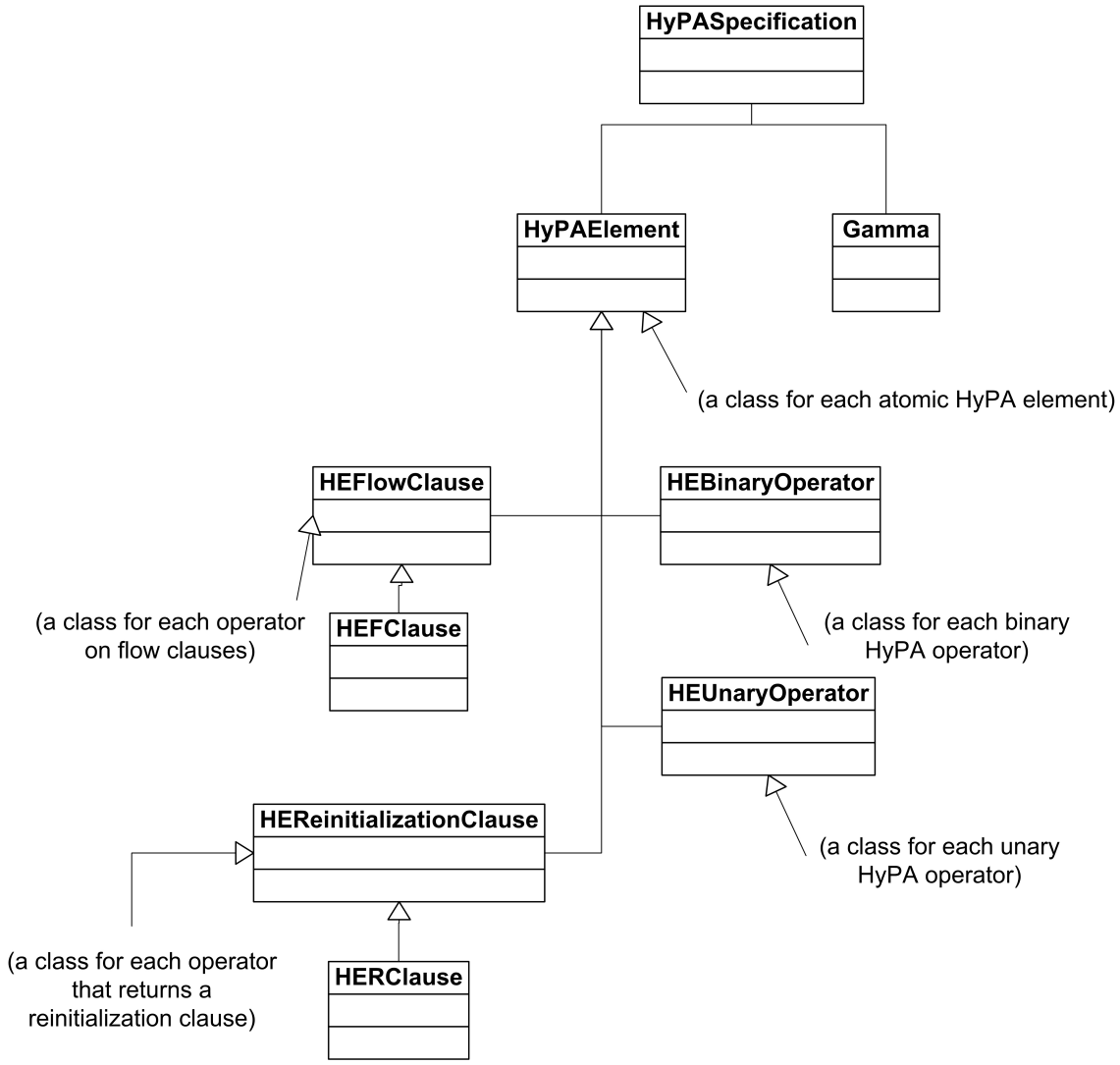
Design decision 5.11 (HyPA element hierarchy)
There is a hierarchy of classes that represent the different HyPA elements (i.e. operators, atomic elements, clauses etc.). The abstract `HyPAElement` class has subclasses for each HyPA element to be represented. Figure 5.10 depicts the class hierarchy for HyPA elements.

There is a subclass for each atomic HyPA element ($\delta,\ \epsilon,\ a,\ X$). Then there are abstract subclasses `HEUnaryOperator` and `HEBinaryOperator` from which classes for all unary and binary operators are derived. Since clauses can consist of atomic clauses and clause operators, they have their own hierarchies: e.g. there is an abstract class `HEReinitializationClause` which has subclasses `HERClause` (atomic clause), `HERUnaryOperator` (from which $^?$ is derived), `HERBinaryOperator` (for $\wedge$ and $\vee$ and $\sim$), and `HERJump` (c_{jmp}).

Design decision 5.12 (Recursive functions)
Recursive functions on process terms, like generating possible first transitions and creating a string representation, are distributed over the HyPA element classes.

Design decision 5.13 (Visitors)
Other operations on process terms can be implemented using so-called Visitors [8]. An abstract class `HyPAVisitor` is defined that is capable of traversing a process-term tree. Subclasses implement some behavior on encountering the different elements. An example of a visitor is the `HyPAActionChecker` for checking the validity of action names in a term.

Figure 5.10: HyPA specification object hierarchy



5.1.3.4 Solutions, times and values

The representation of solutions, times and values depends highly on the clause formalism used. For example, the numerical mathematica formalism uses real numbers while the symbolic mathematica formalism uses symbolic expressions. However, the user interface needs to make some basic assumptions about them in order to be able to draw plots and accept user input:

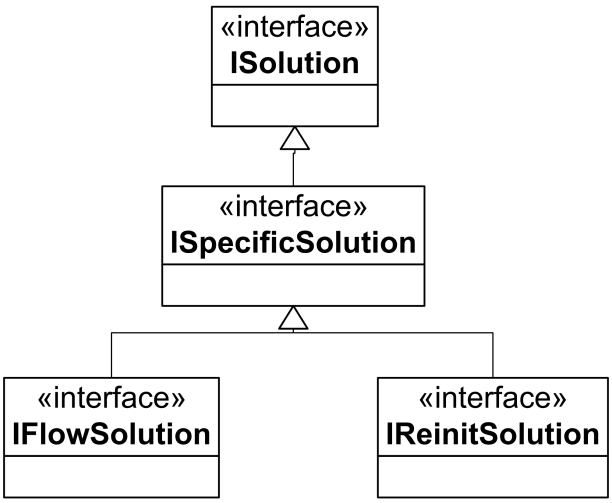
- Times can be converted to and from a positive real number;
- There is a reasonable conversion of values to a real number (for representation in plots);

Design decision 5.14 (`ITime` and `IValue`)
The `ITime` and `IValue` interfaces capture these assumptions. Clause formalisms use different implementations of these interfaces.

For solutions, an interface is needed for the user interface to retrieve values, flows, and maximum time lengths. Also, solutions can be *underspecific* when the behavior of a model variable is not sufficiently constrained by the clause predicates (see also section 5.1.3.2 on "additional predicates"). This has to be communicated to the user.

Design decision 5.15
There is a hierarchy of interfaces for solutions to the different types of clauses, as outlined in figure 5.11. The `ISolution` interface represents underspecific solutions and is little more than a description. The other interfaces deal with specific solutions and they specify methods for retrieving values to plot, and maximum time lengths in case of flow solutions.

Figure 5.11: Solution interface hierarchy



5.1.3.5 Factory

As it turns out, each clause formalism needs its own subclasses of transitions, clauses, values and times. However, these are instantiated in other parts of the simulator than the clause formalism part (see 5.1.4.4). Therefore we use the "Factory class" pattern [8].

Design decision 5.16 (Abstract Factory class)
Each clause formalism has its own `Factory` subclass. The whole application can access this class to create transitions, clauses, values and times specific to the current clause formalism.

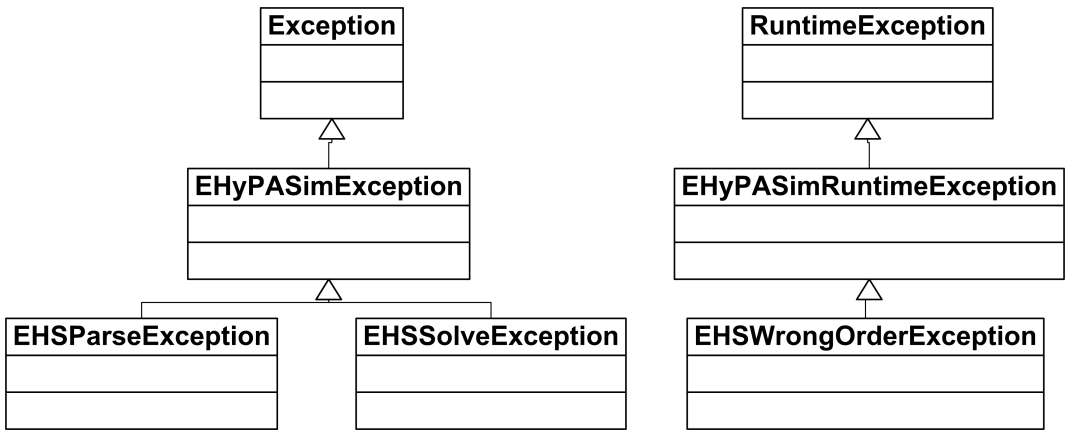
5.1.3.6 Exceptions

A couple of exception classes were created to distinguish simulator exceptions from regular Java exceptions. They are listed in table 5.3 and figure 5.12. As with Java, exceptions are divided into regular and run-time exceptions. Regular exceptions must be caught and handled by the application. Run-time exceptions, mainly method precondition checks, are not expected to occur and are not handled by the application.

Table 5.3: Exceptions

EHyPASimException	Base class for all simulator exceptions.
EHyPASimRuntimeExceptions	Base class for all simulator run-time exceptions.
EHSParseException	Parsing exception (syntax errors, undeclared actions etc).
EHSSolveException	Exception thrown by the clause formalisms.
EHSWrongOrderException	Thrown when methods are called when they should not be.

Figure 5.12: Exceptions hierarchy



5.1.4 Engine

The engine has the tasks of parsing input files, calculating first transitions, and solving clauses. The division of the engine into sub-packages and classes is based on these tasks, and on the extensibility requirements. The engine needs to be extensible in several respects. Requirement

2 states that it should be simple to add another input file format. Requirements 4, 19 and 23 require extensibility with respect to the clause formalism.

Figure 5.13: Engine static structure

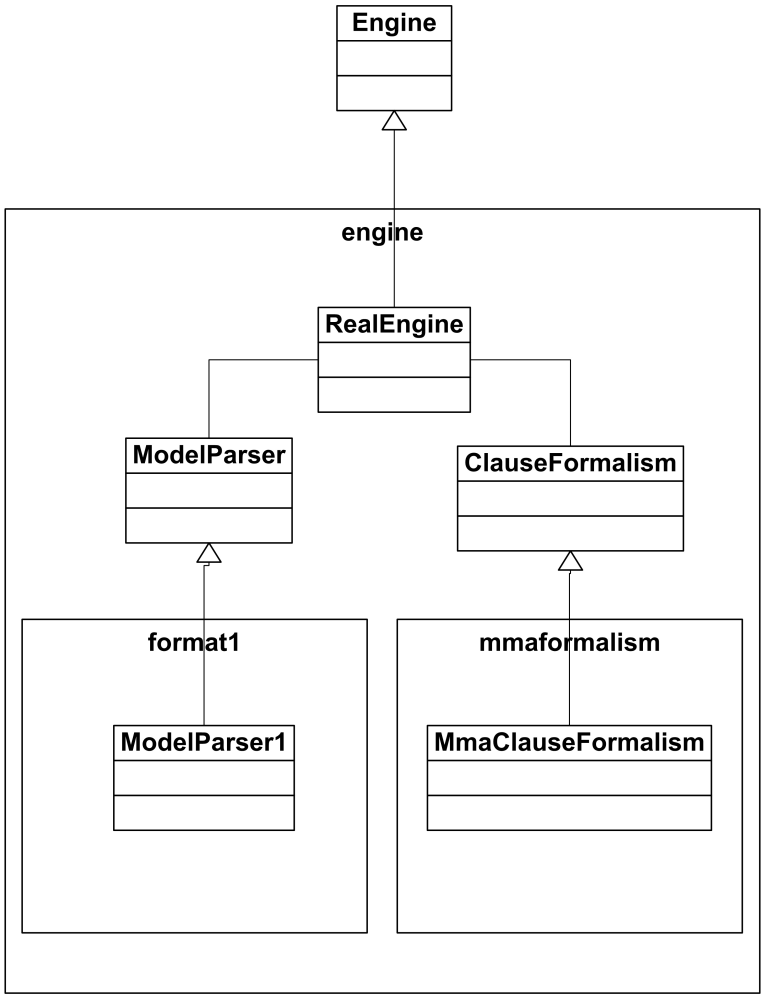


Figure 5.13 shows the structure of the engine. The `RealEngine` façade class implements the `Engine` interface. It uses instances of two abstract classes `ModelParser` and `ClauseFormalism` to do the calculations.

5.1.4.1 State

The `RealEngine` class records the current and previous states. These states are represented by the `State` class and consist of the current specification, process term, valuation, and the current time. The terms are represented by trees of `HyPAElement` objects.

5.1.4.2 Parsing

The simulator has to be extensible with new input file formats.

Design decision 5.17 (Abstract parser class)
There is an abstract `ModelParser` class that specifies a parser. The parser takes a file and returns a `HyPASpecification` object. This HyPA specification consists of the process equations, the initial term, the communication function γ etc. All terms are represented as a tree of `HyPAElement` instances.

For each different input file format, there is to be a subclass of `ModelParser`. Currently there is only one format, "format number 1", described in section 5.2. Its associated subclass is called `ModelParser1`. As it turns out, the input file format is ambiguous. In particular, the parser cannot determine in one pass the difference between flow and reinitialization clauses, and between actions and recursion variables. Therefore, this parser has two stages: in the first stage, a simple tree derived from `HE1Element` is created. In the second stage, the required `HyPAElement` tree is created. The `HE1Element` subclasses are listed in table 5.4.

Table 5.4: HE1Element subclasses

HE1Action	An action or recursion variable;
HE1ActionRename	Action renaming operator;
HE1Atomic	δ or ϵ;
HE1Binary	Any binary operator;
HE1Clause	Any clause;
HE1Encapsulation	$\partial_H(\,)$;
HE1Unary	$^?$ or c_{jmp};
HE1VarRename	Model variable renaming.

5.1.4.3 First transitions

As stated before, the first transitions function is distributed over the `HyPAElement` object tree. Each HyPA element has a method `getFirstTransitions` that implements its own rule of the first transitions function.

Design decision 5.18 (First transitions set)
To store the possible transitions, a class `FirstTransitionsSet` is created. It stores the different types of first transitions (termination, action transition, flow transition) separately so that transitions of a type can be easily retrieved and enumerated.

5.1.4.4 Clause formalism

Design decision 5.19 (Clause formalism Façade)
The `ClauseFormalism` class is a Façade, just like `Engine`. For each different clause formalism, there is a subclass.

The most important methods of the `ClauseFormalism` class are in table 5.5. The main purpose of the `ClauseFormalism` is to solve clauses. The `solve` method does this. It takes a `Transition` object and adds its solutions to it. Each subclass overrides this method. The `getClauseFormalism` method is static. Given the name of a formalism, it returns an instance of the corresponding

Table 5.5: ClauseFormalism methods

getClauseFormalism(name)	Given a clause formalism name, returns a subclass instance;
solve(transition)	Solves the given transition;
getFactory()	Returns a Factory object.

subclass. The method is used during the parsing of an input file to select the proper clause formalism. The `getFactory` method returns a new `Factory` instance for the formalism. The `Factory` class is described in setion 5.1.3.5.

5.2 Input file format

Requirement 1 states that the input file format of the linearization tool must be read by the simulator. However, it turns out that this format does not provide the simulator with all the information that it needs. Therefore we extend the format in the following ways:

Design decision 5.20 (Variable declarations)
Variables need to be declared. There are two reasons for this decision:

1. Declaration of variables gives the simulator the opportunity to check for misspelled variable names in jump sets;
2. In many formalisms, variables cannot be distinguished from constants and functions. To make the distinction, variables need to be declared;

Design decision 5.21 (Right-associative $\gg$ operator)
The reinitialization operator is right-associative, but the linearization tool reads it left-associatively. The simulator follows the HyPA definitions and reads it right-associatively.

Design decision 5.22 (Clause formalism declaration)
The simulator needs to know which clause formalism is used, so there is a clause formalism declaration in every input file.

The consequence of these changes is that models written for the linearization tool must be extended with a clause declaration and variable declarations before they can be simulated. Also, extra brackets may be required to indicate the associativity of re-initializations.

Design decision 5.23 (Syntactic sugar)
To make writing models less cumbersome, some syntactic sugar is added. We mention them here and explain them in detail below.

- Action declarations and communication declarations can be combined into `actcomm` declarations;
- There is an abbreviation `Vm` for the entire set of model variables;
- There is an abbreviation `halves` for all the 'halves' of the communications.

The syntactic sugar described above makes writing models easier, but the linearization tool is unable to read models that use it.

Below, the entire file format is described.

Clause formalism declaration

There is one *clause formalism declaration*, and it must precede the process declarations (described later). It starts with the words `clause formalism` and is followed by the name of the formalism used for the clauses in the model. Currently defined clause formalisms are `mathematica` (numerical mathematica formalism), `symbolicmma` (symbolic mathematica formalism) and `clauseformalismstub` (for testing purposes only).

Example:
`clause formalism mathematica`

Definitions

There may be zero or more *definition declarations*. They start with the word `def`, and between square brackets `[ ]`, global definitions for the clauses (e.g. constants, functions etc.) can be given. These definitions are clause formalism-specific. Examples can therefore be found in section 5.3 that defines the clause formalisms.

Variable declarations

There may be zero or more *variable declarations*. Together they list all used model variables. A variable declaration starts with the keyword `var` followed by one or more variable names. Variables can be separated by whitespace or by commas. Variables consist of a letter followed by letters and digits, and may not be equal to any of the keywords. The used clause formalism may impose further restrictions on the variable names.

Examples:

```
var
    water inflow
    outflow

var
    water,
    inflow, outflow
```

Action declarations

There may be zero or more *action declarations*. Together with the `actcomm` declarations (see below), they list all used discrete actions. An action declaration starts with the keyword `act` followed by one or more action names. Actions can be separated by whitespace or by commas.

Communication declarations

There may be zero or more *communication declarations* that specify the communications function γ. They start with `comm` followed by triples `a|b=c` signifying that $\gamma(a, b) = c$ and $\gamma(b, a) = c$. Triples may be separated by commas or whitespace.

Examples:

```
comm
    ropen | sopen = open
    rclose | sclose = close

comm a|a=b, a|b=c
```

Combined action/communication declarations

Action declarations and communication declarations can become cumbersome to write. Therefore, the keyword `actcomm` is introduced which allows you to declare actions and their communications in one statement (of which you may write zero or more in your model):

Examples:

```
actcomm
    ropen | sopen = open
    rclose | sclose = close

actcomm a|a=b, a|b=c
```

This declares the actions *ropen*, *sopen*, *open*, ... as well as the communications $\gamma(ropen, sopen) = open$ etc.

Process equations

Each process definition $X : p$ is described by a section of the form `proc X = p`, where `X` is the name of a recursion variable and `p` a process term.

Example:

```
proc X = a.X + b.Y
```

Initial process

There is one *initial process term* defined by `initial proc X = p`. In accordance with the file format of Requirement 1, this defines the initial term `p` as well as the definition $X : p$.

Terms

The mapping of HyPA constructs to ASCII constructs is described in table 5.6. Note that brackets `( )` are used solely for grouping, not for flow clauses. Operator precedences are (from strong to weak): $\odot$, $\blacktriangleright$, $\rhd$, $\gg$, $\parallel$, $\between$, $|$, $\oplus$, $\wedge$ and $\vee$, $\sim$, c_{jmp} and $^?$, `[`, `(`. All operators are read left-associatively except for $\odot$ and $\gg$, which are read right-associatively.

For variable renaming $\varrho_{\{x \to y,\ y \to x\}}(p)$, right arrows (`x->y, y->x`) as well as bidirectional arrows (`x<->y`) may be used.

Table 5.6: HyPA to ASCII translation

Description	**HyPA construct**	**ASCII construct**
Deadlock	δ	`delta`
Empty process	ϵ	`epsilon`
Sequential composition	$\odot$	`.`
Disrupt	$\blacktriangleright$	`|>`
Left-disrupt	$\triangleright$	`L>`
Reinitialization	$\gg$	`>>`
Parallel composition	$\parallel$	`||`
Left-parallel composition	$\lfloor\!\lfloor$	`|L`
Forced-synchronization	$\mid$	`|`
Alternative composition	$\oplus$	`+`
Encapsulation	$\partial_{\{a,b,c\}}(p)$	`encap({a, b, c}, p)`
Flow clause	$(\ \{u,v,w\} \mid \mathit{predicate}\ )$	`[ u, v, w | predicate ]`
Flow-and	$\wedge$	$/\backslash$
Reinitialization clause	$[\ \{u,v,w\} \mid \mathit{predicate}\ ]$	`[ u, v, w | predicate ]`
Reinitialization-and	$\wedge$	$/\backslash$
Reinitialization-or	$\vee$	$\backslash/$
Reinitialization-concatenation	$\sim$	`~`
Reinitialization-satisfiable	$?$	`?`
c_{jmp}	$(\ \{u,v,w\} \mid \mathit{predicate}\ )_{jmp}$	`[ u, v, w | predicate ]jmp`
Action renaming	$\rho_{\{a\to b,\ b\to c\}}(p)$	`rename({a->b, b->c}, p)`
Model variable renaming	$\varrho_{\{x\to y,\ y\to x\}}(p)$	`varrename({x<->y}, p), or varrename({x->y, y->x}, p)`

Abbreviations: `Vm` and `halves`

A modeler often needs all model variables in the jump set of a clause. The abbreviation `Vm` may be used for this purpose.

Example:

```
proc X = [ Vm | {speed'[t] == 1, True} ]
```

Usually, all actions that make up the halves of a communication are encapsulated by the ∂_H () operator. This may be specified by the abbreviation `halves`, as shown here:

```
actcomm
    ropen | sopen = open
    rclose | sclose = close

initial proc I = encap({halves}, X || Y || Z)
```

This encapsulates the actions *ropen*, *sopen*, *rclose*, *sclose*.

General restrictions

Names of actions, model variables and recursion variables must be unique. They consist of a letter followed by zero or more letters or digits, and may not be equal to any of the reserved words `clause`, `formalism`, `act`, `comm`, `actcomm`, `var`, `def`, `initial`, `proc`, `delta`, `epsilon`, `rename`, `varrename`, `Vm`, `halves`. The clause formalism that is used may impose further restrictions on names of model variables and constants.

A clause predicate or a definition may be any string as long as all opening and closing square brackets `[ ]` are matched.

Comments and whitespace

Comments start with `%` and stop at the end of the line. Comments may not occur within clauses or definitions. White space is a space, tab, newline or a comment.

Example HyPA model

Table 5.7 gives an example of a model in the input file format.

5.3 Clause formalisms

In the course of the development of the simulator, several clause formalisms are created. After the prototypes, we design "clause formalism number 1" as a first attempt. In order to satisfy modularity requirement 4, this is a self-conceived language that is not solver-dependent. After this turns out to be cumbersome to implement and restrictive to the user, the Mathematica-specific "mathematica" clause formalism is designed. However, numerical computation turns out to have its disadvantages, so the symbolic "symbolicmma" formalism succeeds "mathematica". The following sections describe each formalism.

Table 5.7: Example model in input format

```
clause formalism
    mathematica

var
    waterlevel
    inflow

act
    ropen sopen open
    rclose sclose close

comm
    ropen|sopen=open
    rclose|sclose=close

initial proc I = encap({ropen, sopen, rclose, sclose}, Water || Valve || Controller)

proc Water = [ waterlevel | waterlevel'[t] == inflow[t] - 1 ]

proc Valve = [ inflow | inflow'[t] == 0 ] |>
    ([ inflow | inflowN == 0 ] >> rclose.Valve
    +[ inflow | inflowN == 3 ] >> ropen.Valve)

proc Controller = [ | True ] |> (ropen.Controller + rclose.Controller)
```

5.3.1 Formalism number one

Formalism number one was never fully implemented, and is not supported anymore by the final versions of the simulator. Therefore we do not describe it in detail. Its language was designed with the thought that we wanted to get something working soon, and not with any practical application in mind. It is a language consisting of variables, functions, the basic operators $+ - / *$, remainder, power, ln, log. This language was then to be translated to the different mathematical solvers (mathematica, matlab etc.). A huge drawback is the difficulties this creates in representing the answers returned by the solver, since the set of operators is limited.

5.3.2 Mathematica clause formalism

The "mathematica" clause formalism is meant to give the user the freedom to specify most of what Mathematica allows. Predicates consist of (differential) equations and inequalities, in Mathematica notation. Numerical methods are used to solve the clauses. This decision is backed by the fact that more differential equations are solvable by numerical methods than by symbolic methods.

Identifiers

Variables and constants may have any name that Mathematica allows, except for `t`. The name `t` is used as a independent (time) variable for the differential equations. Using lowercase letters ensures that a name does not coincide with an existing Mathematica symbol.

Definitions

Constants can be defined in a `def` section. They are given as `<name> = <value>` constructs separated by semicolons. The names must be valid, unique identifiers, and the values must be Mathematica Real literals.

Example:

```
def [ wMin = 3.; wMax = 6.0; ]
```

Flow predicates

Flow predicates consist of:

- (differential) equations that specify continuous behavior over time;
  - The symbol `==` is used to specify an equation.
  - Differential equations must be stated in the form that is understood by the Mathematica `NDSolve` function.
  - Initial and boundary conditions are typically stated in form `y[`t_0`] ==` c_0`, y'[`t_0`] ==` dc_0, etc., but may consist of more complicated equations.
  - There must be exactly enough initial conditions to specify a single particular solution. Initial conditions are also introduced by adding model variables to the jump set of the flow clause.
- Inequalities that specify conditions on the continuous behavior e.g. $x < 13$. These inequalities are solved by shortening the time domain of the solution to the equalities.
  - The symbols `< > <= >=` can be used to specify inequalities.
  - Conditions on the solutions of the equalities can be stated as e.g. `x < 100` which is taken to mean that the solution for `x[t]` must be less than `100` on its entire domain. The simulator accomplishes such a condition by reducing the domain.

- Equations and inequalities are combined using `&&`.
- All model variables are treated as functions over `t`, their domain ranging up from 0. No model variable may have name `t`.
- All existing Mathematica functions and constants may be used, although `NDSolve` does not find solutions for every possible differential equation. Notably, the `Sign` function causes Mathematica 5.1 to crash when used within a `NDSolve` call.
- The predicates `False` and `True` may be used on their own as flow predicates.

A flow $\sigma : \mathcal{V}_\mathrm{m} \to [0 \ldots t] \to \mathbb{R}$ is a solution to a flow clause if the equalities hold on the entire domain $[0 \ldots t]$ and the inequalities on the open domain $[0 \ldots t)$.

Example: `x'[t] == x[t] + y[t] && y[t] == 2 * Sin[t] && y < Pi`

Reinitialization predicates

Reinitialization predicates are combinations of equalities and inequalities over the previous and next value of the model variables.

- Equations and inequalities are combined using `&&`.
- The symbol `==` is used to specify an equality.
- The symbols `< > <= >=` can be used to specify inequalities.
- A previous value is indicated by the name of the model variable followed by the letter `P`, and a next value is indicated by the name of the model variable followed by the letter `N`. All model variable occurrences *must* have suffix `P` or `N`.
- The predicates `False` and `True` may be used.

Example: `xP < 5 && xN == xP + 10`

Be cautious with equalities in reinitialization predicates. Due to the errors that occur in numeric calculation, variables most likely do not reach a specified exact value. For example, the process term

$$\left(\; x \;\middle|\; \mathtt{x'[t]} == 1 \;\&\&\; \mathtt{x} < 4 \; \right) \rhd \left[\middle| \; \mathtt{xP} == 4 \; \right] \gg \epsilon$$

will most likely not terminate successfully.

A solution to this problem could be to interpret `a == b` as $|a - b| < e$ for some user-specified constant e. However, this interpretation incurs its own problems. Suppose there are 'previous' as well as 'next' values in an equation, e.g. `aP == aN`. This new interpretation leads to many possible solutions for the reinitialization clause. Another problem with the approach are constructs of the form $\left[\middle| \; \mathtt{xP} == 0 \; \right] \gg \left(\; x \;\middle|\; \mathtt{x[t]} == 0 \; \right)$. If `xP == 0` is interpreted as $|x| < e$ then a solution for the flow clause is not likely to be found (only in the case where `x` is exactly zero).

5.3.3 Symbolic Mathematica clause formalism

The numerical "mathematica" formalism has its problems:

- Constructions like $\left(\ V \ \middle| \ x'(t) = 1 \wedge x(t) < 3 \ \right) \blacktriangleright \left(\left[\ x^- = 3 \ \right] \gg a \right)$ pose a problem, since the exact value 3 is probably never reached (due to calculation errors). This leads to deadlocks in the simulation where there would not be any in analysis;
- Sometimes the Mathematica NMinimize function returns an answer from which we cannot deduct wether the inequalities hold on the entire domain or not at all. This happens for example when $\sigma(t)(x)' = 0$;
- The accuracy and precision of the calculation cannot really be determined by the user.

Therefore we introduce a new "symbolicmma" clause formalism that solves its clauses symbolically.

Identifiers

Variables, constants and functions may have any name that Mathematica allows, except for `t` and `<var><suf><n>`. Here `<var>` is a variable name, `<suf>` is `P` or `N`, and `<n>` is a number.

The name `t` is used as a independent (time) variable for the differential equations. The other type of disallowed identifier is used for temporary variable names during the solving process.

Using lowercase letters ensures that a name does not coincide with an existing Mathematica symbol.

Definitions

In each `def[ ... ]` section, mathematica commands may be listed that are executed prior to solving a transition. Commands must be separated by semicolons. This allows for example for the definition of constants and functions. All expressions must be symbolic. This implies that numbers with decimal points are not allowed.

Example:

```
def [ wMin = 3; wMax = 60; distance[x1_, x2_] := Sqrt[(x2 - x1)^ 2]; ]
```

Flow clauses

A flow predicate is either `False`, `True`, or a list of length 2.

The list is enclosed by { }. The first element consists of (differential) equations, combined using `&&`. The second element is any logical combination of inequalities.

The equations are solved by the `DSolve` function. Inequalities are solved by `Reduce[..., t, Reals]`. All model variable occurrences must range over variable `t`.

Here is the formal definition of a solution to a flow clause:

Definition 5.1 ($\models_{\mathrm{f}}$)
Let σ be a continuous flow with domain $[0 \ldots t]$ for some t. Let $P_{\mathrm{f}} = \{E,\ I\}$ be a valid flow predicate over the model variables over time. We define $\models_{\mathrm{f}}$ as follows:

$$\sigma \models_{\mathrm{f}} P_{\mathrm{f}} \text{ iff } \forall_{t' \in [0 \ldots t]} E(t') \text{ and } \forall_{t' \in [0 \ldots t)} I(t')$$

Note that on the upper border of the domain of σ, the inequalities do not have to hold! This ensures that constructions like $\left(\ V \ \middle| \ x'(t) = 1 \wedge x(t) < 3 \ \right) \blacktriangleright \left(\left[\ x^- = 3 \ \right] \gg a \right)$ are possible.

Examples:
{x'[t] == x[t] + y[t] && y[t] == 2*Sin[t], (y[t] < Pi || y[t] > Pi)}

{water'[t] = inflow[t] - outflow[t], True}

Reinitialization clauses

Reinitialization predicates are logical combinations of equalities and inequalities over the previous and next value of the model variables. A previous value is indicated by the name of the model variable followed by the letter `P`, and a next value is indicated by the name of the model variable followed by the letter `N`. All model variable occurrences *must* have suffix `P` or `N`. The predicates `False` and `True` may be used.

Example: `xP < 5 && xN == xP + 10`

Chapter 6

Implementation

This chapter describes decisions made in the Implementation phase. Detailed design documentation is provided in comments in the code itself. The comments use JavaDoc syntax so that documentation in HTML form is easily generated.

6.1 Programming language

We choose Java as a programming language, for a number of reasons:

- Java is the only language with which all of the considered solver tools (Matlab, Maple, Mathematica) can communicate;
- Performance is expected to be more of an issue of the mathematical solver than of the simulator;
- Java offers garbage collection and compile-time checks of exception declaration, taking some concerns out of our hands.

JBuilder 2005 Enterprise is used as development environment, since it is available at the TU/e.

6.2 Coding style

We provide JavaDoc comments for obvious reasons. We adhere to most of the coding guidelines that can be checked by the JBuilder Coding Audits. They seem reasonable and can obviously be checked automatically. JBuilder automatically takes care of formatting.

6.3 Parsing

The well-known combination of JFLEX and JavaCUP is used to generate the various parsers needed by the simulator. Separate parsers are used for the HyPA language, and, per clause formalism, for reinitialization and for flow clauses. The grammars for the input languages can be found in appendix G.

6.4 Plotting

Inspired by the simulators cHARON and HyVisual, we started out by using the Ptolemy Plot library [13] for plotting continuous behavior. Extending these classes proved difficult, and we designed our own plot classes specifically for this solver (see also section 5.1.2.1).

Chapter 7

Cases

This chapter lists the more interesting HyPA models used in case studies. Many smaller test models are not listed in this document. All models described here are simulated using the symbolic mathematica clause formalism, althogh for most models, numeric versions were also constructed. For readability and for consitency throughout this document, all models are not given in the input file format but in standard HyPA format.

7.1 Steam boiler

The model in table 7.1 is the steam boiler example from [7, section 2.4]. There is a container of water. The water level must be kept between a minimum and maximum level. Water continuously evaporates from the container. A controller has the ability to open or close a valve, which releases water into the container. The controller looks at the current water level on a regular basis and opens or closes the valve depending on the measured water level.

Some changes were made to the model after taking it from [7]. The "nothing" action is introduced to make the model substitutably guarded. Furthermore, the behavior of "steam" is now specified. This lessens the interaction that is needed between user and simulator during simulation.

The results of a simulation session with the boiler model can be seen in figure 7.1. The top plot shows the inflow (the bottom line) and the water level (the saw-tooth shaped top line). The bottom plot shows the controller clock. You can see by the "open", "close" and "nothing" actions that the controller makes a decision each time the clock reaches the value of "clockPeriod".

The simulator shows a shortcoming of the model itself. Dependent on its initial value, the water level can become negative. This shows that the simulator can indeed be useful to correct some obvious mistakes in a model before the application of formal analysis tools.

Because this was the first model to be simulated, it has served mainly as a test for the user interface. We adapted colors, font sizes, placements of value labels on plot axes etc. according to experiences with this model.

Also, the model shows that the choice of solving clauses numerically or symbolically impacts the model itself. For numeric calculation, the construction

$$\left(\text{ clock } \middle| \quad \text{clock}'(t) = 1,\ \text{clock}(t) < \text{clockPeriod} \quad \right) \blacktriangleright \left[\middle| \quad \text{clock}^{-} = \text{clockPeriod} \quad \right] \gg \ldots$$

does not work: the value of "clockPeriod" is probably never reached due to the errors introduced by numeric calculation. This means that while the model does not have a deadlock, it does during numeric simulation. Therefore, one has to adapt one's model to numeric calculation by eliminating equalities.

Table 7.1: Steam boiler model

$\mathcal{A} = \{\text{close},\ \text{nothing},\ \text{open},\ \text{rclose},\ \text{ropen},\ \text{sclose},\ \text{sopen}\}$

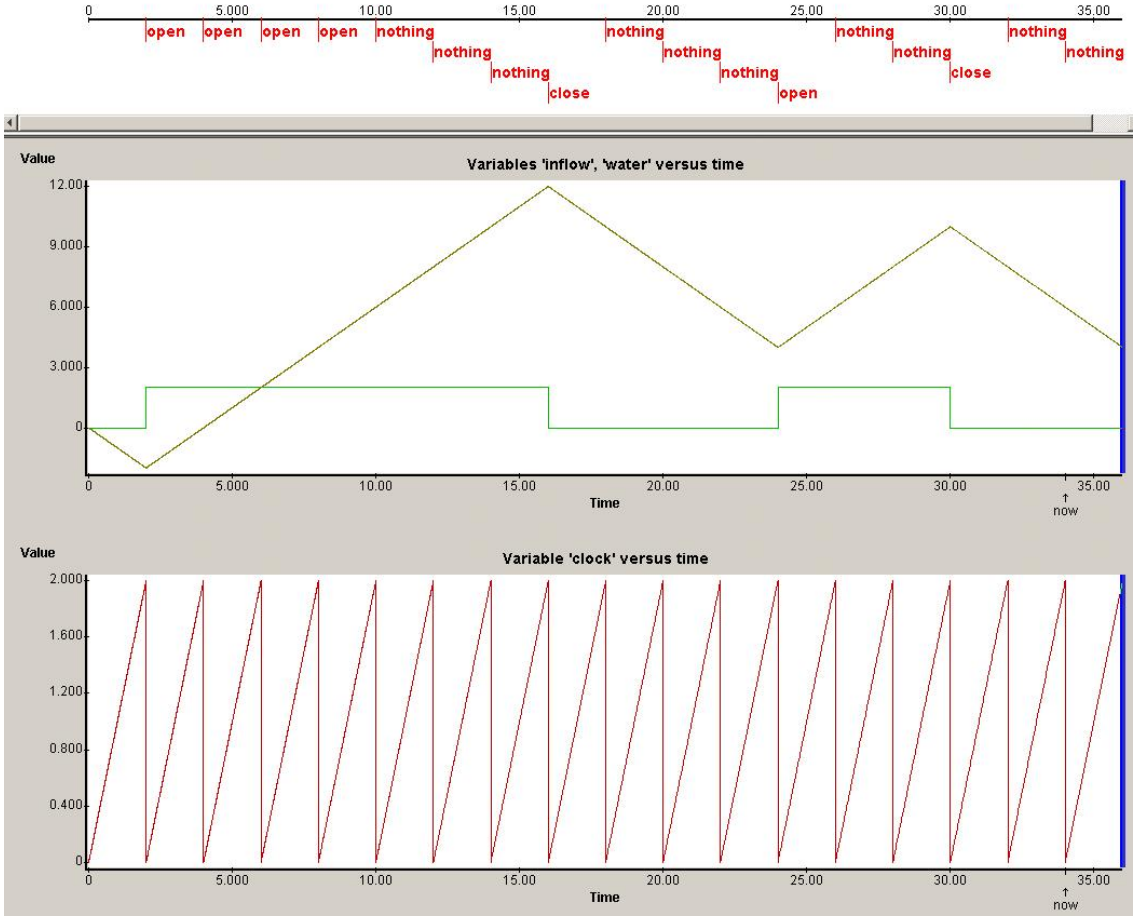
$\gamma(\text{sclose}, \text{rclose}) = \text{close},\ \gamma(\text{sopen}, \text{ropen}) = \text{open}$

$\mathcal{V}_{\text{m}} = \{\text{clock},\ \text{inflow},\ \text{steam},\ \text{water}\}$

$\text{steamOut} = 1;\ \text{inflowMax} = 2;\ \text{clockPeriod} = 2;\ \text{waterSafeMin} = 5;\ \text{waterSafeMax} = 10;$

$$
\begin{array}{lll}
\textbf{Boiler} & : & \partial_{\text{rclose, ropen, sclose, sopen}}\,(\text{Water} \parallel \text{Heater} \parallel (\text{ValveOpen} \oplus \text{ValveClosed}) \parallel \text{Controller}) \\
\text{Controller} & : & \\
 & & \quad [\ \text{clock} \mid \text{clock}^{+} = 0\ ] \gg ((\ \text{clock} \mid \text{clock}'(t) = 1,\ \text{clock}(t) < \text{clockPeriod}\ ) \blacktriangleright \\
 & & \qquad [\mid \text{clock}^{-} = \text{clockPeriod}\ ] \gg \\
 & & \qquad (\\
 & & \qquad\quad [\mid \text{water}^{-} \geq \text{waterSafeMax}\ ] \gg (\text{sclose} \odot \text{Controller}) \\
 & & \qquad \oplus \\
 & & \qquad\quad [\mid \text{waterSafeMin} \leq \text{water}^{-} \leq \text{waterSafeMax}\ ] \gg (\text{nothing} \odot \text{Controller}) \\
 & & \qquad \oplus \\
 & & \qquad\quad [\mid \text{water}^{-} \leq \text{waterSafeMin}\ ] \gg (\text{sopen} \odot \text{Controller}) \\
 & & \qquad) \\
 & & \quad) \\
\text{Heater} & : & (\mid \text{steam}(t) = \text{steamOut}\ ) \\
\text{ValveClosed} & : & (\mid \text{inflow}(t) = 0\ ) \blacktriangleright ((\text{rclose} \odot \text{ValveClosed}) \oplus (\text{ropen} \odot \text{ValveOpen})) \\
\text{ValveOpen} & : & (\mid \text{inflow}(t) = \text{inflowMax}\ ) \blacktriangleright ((\text{rclose} \odot \text{ValveClosed}) \oplus (\text{ropen} \odot \text{ValveOpen})) \\
\text{Water} & : & (\ \text{water} \mid \text{water}'(t) = \text{inflow}(t) - \text{steam}(t)\ )
\end{array}
$$

Figure 7.1: Boiler simulation results



7.2 Thermostat

The thermostat (table 7.2), taken from [5], is a small example to show that our plots can display curves nicely. There is a heater that is supposed to keep the temperature in a room between two limits. To do so, it switches on and off upon reaching temperatures $\mathrm{T_{min}}$ and $\mathrm{T_{max}}$, respectively. Two constants K and h influence the rate at which warmth leaks from the room and the amount of warmth generated by the heater. The results are in figure 7.2 (although this figure is a bit blurred). Zooming in works fine and leaves the curves intact.

Table 7.2: Thermostat model

$\mathcal{A} = \{\mathrm{off},\ \mathrm{on}\}$

$\mathcal{V}_\mathrm{m} = \{\mathrm{T}\}$

$\mathrm{K} = 8 \times 10^{-1};\ \mathrm{h} = 50;\ \mathrm{T_{min}} = 10;\ \mathrm{T_{max}} = 30;$

$$\textbf{Thermostat} \quad : \quad \mathrm{ThermostatOn} \oplus \mathrm{ThermostatOff}$$

$$\mathrm{ThermostatOff} \quad : \quad \left(\mathrm{T} \;\middle|\; \mathrm{T}'(t) = -\mathrm{K} \times \mathrm{T}(t),\ \mathrm{T}(t) > \mathrm{T_{min}} \right) \blacktriangleright \left[\middle| \; \mathrm{T}^- = \mathrm{T_{min}} \right] \gg \mathrm{on} \odot \mathrm{ThermostatOn}$$

$$\mathrm{ThermostatOn} \quad : \quad \left(\mathrm{T} \;\middle|\; \mathrm{T}'(t) = \mathrm{K} \times (\mathrm{h} - \mathrm{T}(t)),\ \mathrm{T}(t) < \mathrm{T_{max}} \right) \blacktriangleright \left[\middle| \; \mathrm{T}^- = \mathrm{T_{max}} \right] \gg \mathrm{off} \odot \mathrm{ThermostatOff}$$

7.3 Bouncing ball

The Bouncing Ball example from [5] is a nice model because it has very small time steps. The model is in table 7.3, and the simulation results in figure 7.3. There is a ball that is dropped from a certain height. The top plot represents the altitude of the ball, and the bottom plot shows the speed. Each time the ball hits the ground, it loses some of its energy according to a factor *f*. However, the model is such that the ball never stops bouncing, the bounces only become smaller and smaller. For the simulator this means that the length of the transitions becomes shorter and shorter.

For numeric simulation, this model is difficult, because at some point, the time lengths become shorter than the presicion/accuracy of the calculation. In practice, the ball eventually bounces to the same height every time.

7.4 Billiards

This example is taken from [17]. It shows a ball bouncing around a billiards table. There are variables 'mx' and 'my' representing the size of the table, 'vx' and 'vy' representing ball speed in the two directions, and 'x' and 'y' for ball position. Initially, the ball is somewhere on the table, and it already has some speed.

The model from [17] is extended (table 7.4) so that the ball stops eventually. There is a constant 'f' that models the friction between the ball and the table. This causes the velocity of the ball to drop. When a certain minimum speed is reached, the model goes into a state in which the speed is actually zero.

Figure 7.2: Thermostat simulation results

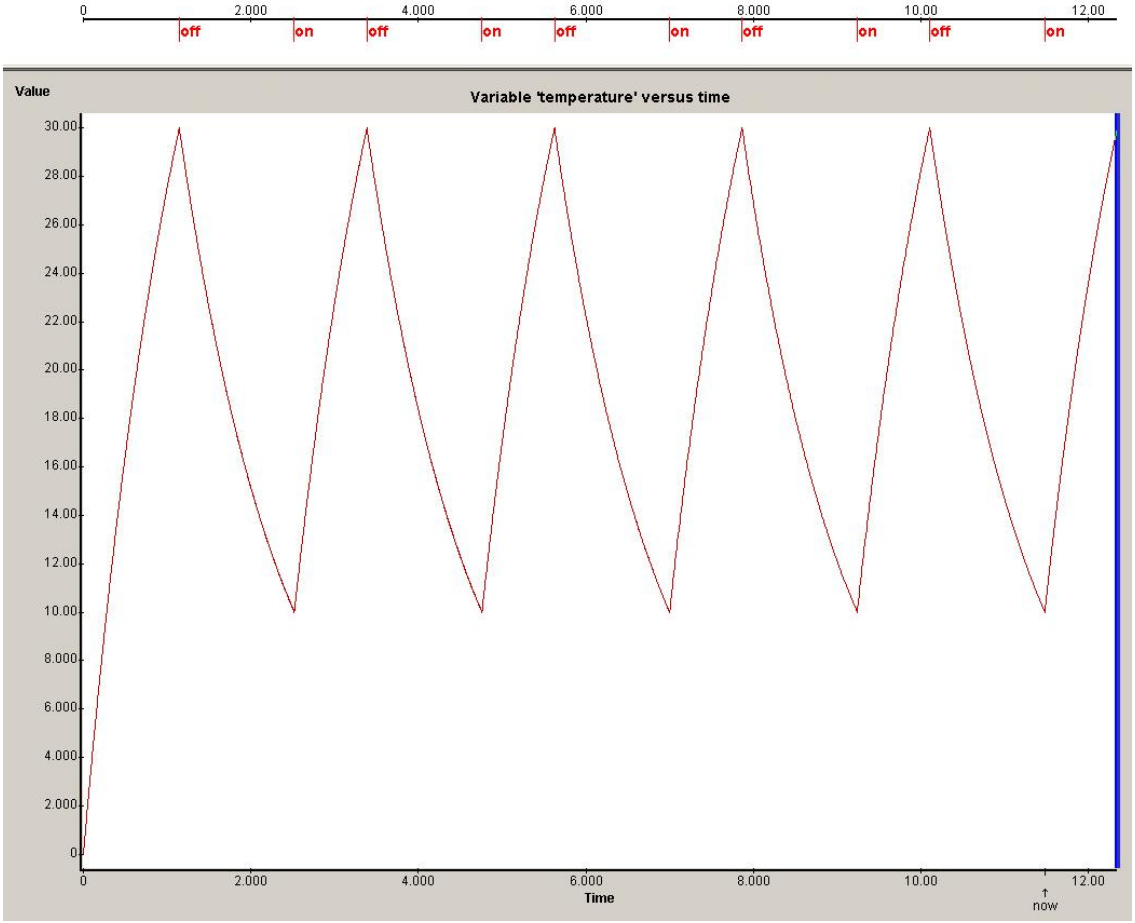


Table 7.3: Bouncing Ball model

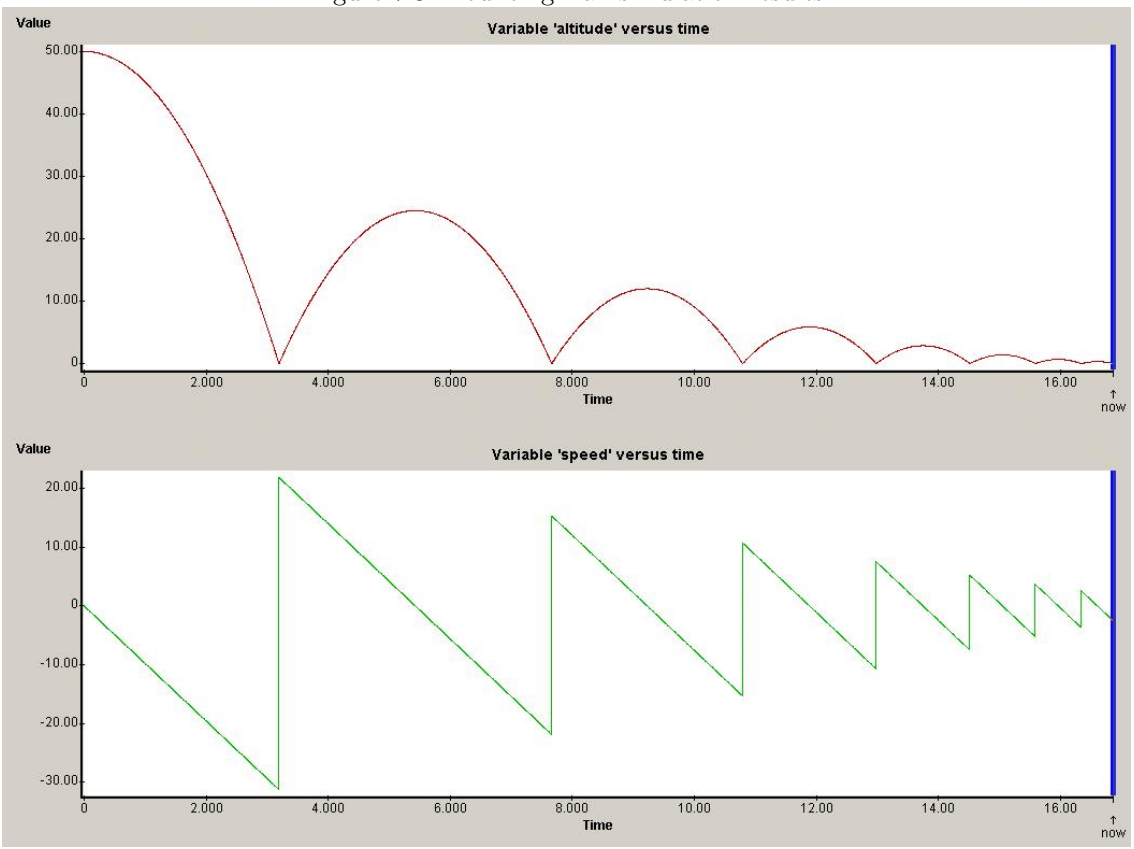
$\mathcal{A} = \{\}$

$\mathcal{V}_{\mathrm{m}} = \{\text{altitude},\ \text{speed}\}$

$g = 981 \times 10^{-2};\ f = 70 \times 10^{-2};$

BouncingBall	:	$\left[\!\middle|\ \text{speed}^- > 0\ \right] \gg \text{Up} \oplus \left[\!\middle|\ \text{speed}^- <= 0\ \right] \gg \text{Down}$
Up	:	$\left(\ \text{altitude},\ \text{speed}\ \middle|\ \text{altitude}'(t) = \text{speed}(t) \wedge \text{speed}'(t) = -1 \times g,\ \text{speed}(t) \geq 0\ \right) \triangleright$ $\left[\!\middle|\ \text{speed}^- = 0\ \right] \gg \text{Down}$
Down	:	$\left(\ \text{altitude},\ \text{speed}\ \middle|\ \text{altitude}'(t) = \text{speed}(t) \wedge \text{speed}'(t) = -1 \times g,\ \text{altitude}(t) \geq 0\ \right) \triangleright$ $\left[\ \text{speed}\ \middle|\ \text{altitude}^- = 0\ \wedge\ \text{speed}^+ = -1 \times f \times \text{speed}^-\ \right] \gg \text{Up}$

Figure 7.3: Bouncing Ball simulation results



This model shows that the model variable renaming operator, introduced in section 4.2, is useful. The calculations for the X and Y directions are the same. We introduce processes that only calculate values for the X-direction. We run that process in parallel with itself, with variables renamed for the Y-direction.

In figure 7.4 there are two plots. The top one shows the speed in X and Y-direction. You can see that the speeds reverse direction on each bounce, and that the speed decreases due to the friction. The bottom one shows the position of the ball. This shows that an X/Y-plot is a natural way to visualize some systems: you can actually see the ball bouncing around the table. The initial kink in the line starting from coordinates $(0, 0)$ is from the reinitialization at the beginning of the model. The simulator had its initial variable values all set to 0, and then the reinitialization placed the ball on coordinates $(5, 2)$.

Table 7.4: Billiards model

$\mathcal{A} = \{\}$

$\mathcal{V}_{\mathrm{m}} = \{\mathrm{mx},\ \mathrm{my},\ \mathrm{vx},\ \mathrm{vy},\ \mathrm{x},\ \mathrm{y}\}$

$\mathrm{f} = -5 \times 10^{-2};\ \mathrm{stopspeed} = 1 \times 10^{-2};$

$$\textbf{Billiards} \ : \ \left[\ \mathrm{mx},\ \mathrm{my},\ \mathrm{vx},\ \mathrm{vy},\ \mathrm{x},\ \mathrm{y}\ \middle|\ \begin{array}{c}\mathrm{x}^{+} = 5\ \wedge\ \mathrm{mx}^{+} = 40\ \wedge\ \mathrm{vx}^{+} = 4\ \wedge \\ \mathrm{y}^{+} = 2\ \wedge\ \mathrm{my}^{+} = 10\ \wedge\ \mathrm{vy}^{+} = 3\end{array}\ \right] \gg$$

$$\mathrm{MovePos} \parallel \varrho_{\mathrm{mx}\leftrightarrow\mathrm{my},\mathrm{vx}\leftrightarrow\mathrm{vy},\mathrm{x}\leftrightarrow\mathrm{y}}(\mathrm{MovePos})$$

$$\mathrm{MoveNeg} \ : \ \left(\ \mathrm{mx},\ \mathrm{vx},\ \mathrm{x}\ \middle|\ \begin{array}{l}\mathrm{x}(t) > 0\ \wedge\ \mathrm{vx}(t) < 0, \\ \mathrm{mx}'(t) = 0\ \wedge\ \mathrm{vx}'(t) = \mathrm{f} * \mathrm{vx}(t)\ \wedge\ \mathrm{x}'(t) = \mathrm{vx}(t)\end{array}\ \right) \blacktriangleright$$

$$\begin{array}{l}\left[\ \middle|\ \ \mathrm{vx}^{-} > -1 * \mathrm{stopspeed}\ \ \right] \gg \mathrm{Still} \\ \oplus \\ \left[\ \mathrm{vx}\ \middle|\ \ \mathrm{x}^{-} = 0\ \wedge\ \mathrm{vx}^{+} = -\mathrm{vx}^{-}\ \ \right] \gg \mathrm{MovePos}\end{array}$$

$$\mathrm{MovePos} \ : \ \left(\ \mathrm{mx},\ \mathrm{vx},\ \mathrm{x}\ \middle|\ \begin{array}{l}\mathrm{x}(t) < \mathrm{mx}(t)\ \wedge\ \mathrm{vx}(t) > 0 \\ \mathrm{mx}'(t) = 0\ \wedge \mathrm{vx}'(t) = \mathrm{f} \times \mathrm{vx}(t)\ \wedge\ \mathrm{x}'(t) = \mathrm{vx}(t)\end{array}\ \right) \blacktriangleright$$

$$\begin{array}{l}\left[\ \middle|\ \ \mathrm{vx}^{-} < \mathrm{stopspeed}\ \ \right] \gg \mathrm{Still} \\ \oplus \\ \left[\ \mathrm{vx}\ \middle|\ \ \mathrm{x}^{-} = \mathrm{mx}^{-}\ \wedge\ \mathrm{vx}^{+} = -\mathrm{vx}^{-}\ \ \right] \gg \mathrm{MoveNeg}\end{array}$$

$$\mathrm{Still} \ : \ \left(\ \mathrm{mx},\ \mathrm{vx},\ \mathrm{x}\ \middle|\ \ \mathrm{mx}'(t) = 0\ \wedge\ \mathrm{vx}'(t) = 0\ \wedge\ \mathrm{x}'(t) = 0\ \ \right)$$

7.5 Simple test

Table 7.5 shows a model that allows the user to freely simulate any behavior. In each step, the user has the choice between an action, a flow (one in which the variable may jump), or successful termination. The flow is not specific, so the user will be asked for a predicate each time.

Figure 7.4: Billiards simulation results

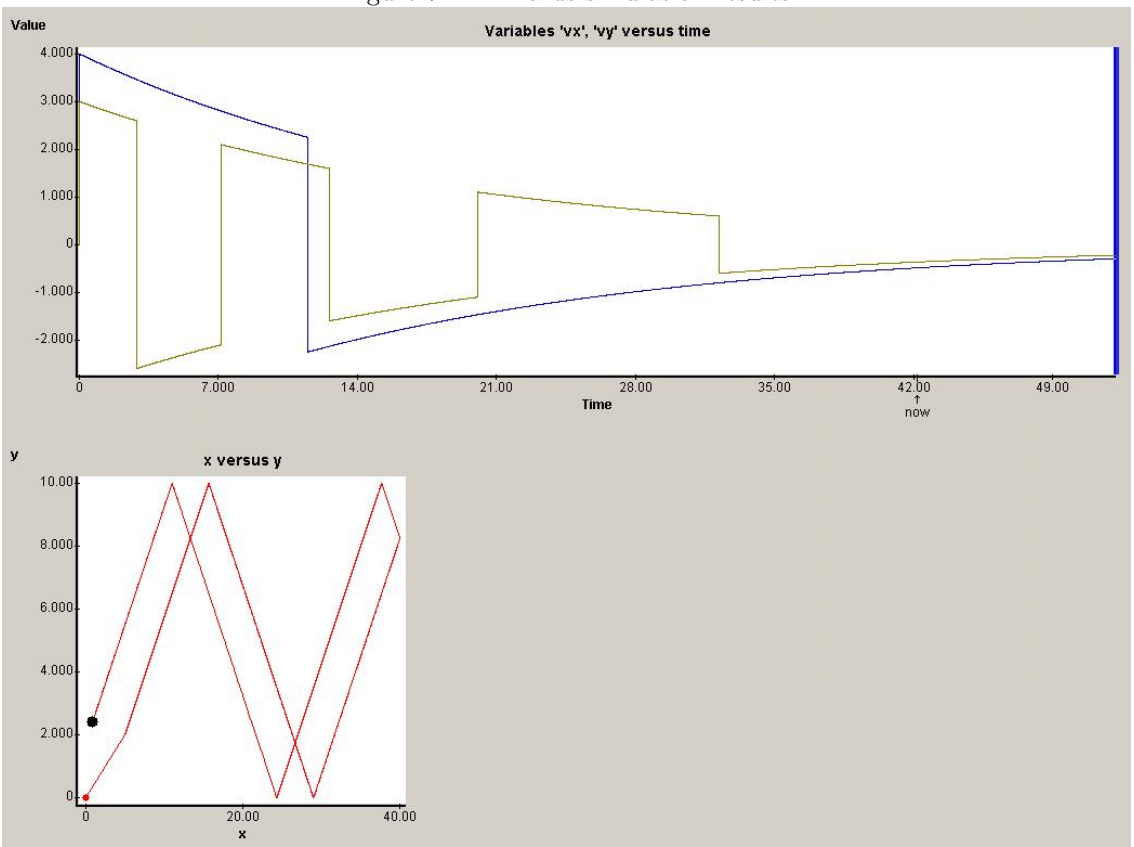


Table 7.5: Simple test model

$\mathcal{A} = \{\mathrm{a}\}$

$\mathcal{V}_{\mathrm{m}} = \{\mathrm{x}\}$

$\mathbf{I} \quad : \quad \mathrm{a} \odot \mathrm{I} \oplus \left(\mid \quad \mathit{True} \quad \right) \blacktriangleright \mathrm{I} \oplus \epsilon$

7.6 Half wave rectifier

Figure 7.5 shows a half wave rectifier circuit, taken from [17]. It consists of a diode D, two resistors with resistance R_0 and R_1, respectively, a capacitor with capacity C_0, and a voltage source with voltage v_0. The source voltage is a sine wave with frequency f.

For each component in the circuit, there is a process in the HyPA model. For the passive components, this is a single flow clause that defines a relation between voltages and currents in the system. For the diode, we introduce two modes of operation: off and on.

We introduce a Clock process that models time, so that we can produce the sine wave using the formula $v_0'(t) = \sin(2\pi f\, clock(t))$.

In this model, there is no clear division of the model variables over the different processes. For example, to which process does variable i_0 belong? Current i_0 is influenced by R_0 as well as by diode D. Therefore, which variables should be in the jump sets of the different flow clauses? Luckily, all variables in this circuit can be modeled as continuous ones, so that $\mathcal{V}_\mathrm{m}$ may be put into all of the jump sets.

Simulating the model turns out to be a challenge. The symbolic clause formalism turns out not to be able to handle the equations:

$$clock(0) = 0 \ \wedge \ clock'(t) = 1 \ \wedge \ v_0(0) = 0 \ \wedge \ v_0'(t) = \sin(2\pi f\, clock(t))$$

The mathematica `DSolve` function simply does not accept them. We replace the sine waveform by a saw-tooth waveform, using the process:

$$\begin{aligned} \text{Source} \ : \ & \big(\ \mathcal{V}_\mathrm{m} \ \big| \ \ v_0'(t) = 1 \ \wedge \ v_0(t) < 1 \ \ \big) \rhd \\ & \big[\big| \ v_0^- = 1 \ \big] \gg \big(\ \mathcal{V}_\mathrm{m} \ \big| \ \ v_0'(t) = -1 \ \wedge \ v_0(t) > -1 \ \ \big) \rhd \\ & \big[\big| \ v_0^- = -1 \ \big] \gg \text{Source} \end{aligned}$$

The resulting model *can* be simulated, and the resulting voltages and currents can be found in figures 7.6 and 7.7.

To simulate the sine wave, numeric calculation must be tried. This requires some rewriting of the model, as the reinitialization predicate contains equalities. We would like to replace the predicate

$$i_0^- = 0 \ \wedge \ v_1^- = v_2^-$$

by something like

$$|i_0^-| < 0.001 \ \wedge \ |v_1^- - v_2^-| < 0.001$$

This creates a problem with the flow clause for DiodeOff: unless i_0 is exactly zero, the equation $i_0(t) = 0$ cannot be satisfied without a jump at the start of the flow. A solution to this is to extend the reinitialization with $i_0^+ = 0 \ \wedge \ v_1^+ = v_2^-$. This ensures that i_0 is zero and that v_1 equals v_2 after the reinitialization. But it creates difficulties with the other differential equations: some variables have jumped and some have not, creating a discrepancy. Now, the only solution is to let the other variables jump as well, in such a fashion that all equations are satisfied at the beginning of the next flow. This means that knowledge about the whole system is now incorporated in the reinitialization for the diode.

Even after rewriting the model in this fashion (table 7.7), it cannot be simulated: the mathematica `NMinimize` function cannot handle inequalities on the sine wave. This implies that the flows do not end when they should, so that the diode does not switch modes in time. The sine wave remains un-simulated.

Figure 7.5: Half wave rectifier (figure taken from [17])

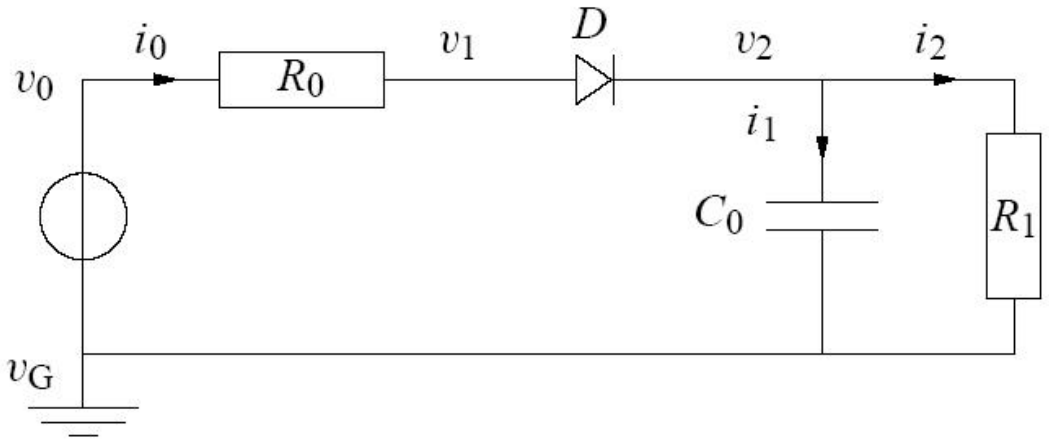


Table 7.6: Rectifier

$\mathcal{V}_m = \{clock,\ i_0,\ i_1,\ i_2,\ v_0,\ v_1,\ v_2\}$

$f = 1;\ R_0 = 10;\ R_1 = 30;\ C_0 = 50 \times 10^{-3};$

Rectifier	:	Clock ‖ Source ‖ Resistor0 ‖ Capacitor0 ‖ Resistor1 ‖ Diode
Capacitor0	:	$(\ \mathcal{V}_m \ \mid \ C_0 v_2'(t) = i_1(t) \ )$
Clock	:	$(\ \mathcal{V}_m \ \mid \ clock'(t) = 1 \ )$
Diode	:	$[\mid \ i_0^- = 0 \ \wedge \ v_1^- = v_2^- \ ] \gg ((\text{DiodeOn} \oplus \text{DiodeOff}) \rhd \text{Diode})$
DiodeOff	:	$(\ \mathcal{V}_m \ \mid \ i_0(t) = 0 \ \wedge \ i_1(t) = -i_2(t) \ \wedge \ v_1(t) \leq v_2(t) \ )$
DiodeOn	:	$(\ \mathcal{V}_m \ \mid \ v_1(t) = v_2(t) \ \wedge \ i_0(t) = i_1(t) + i_2(t) \ \wedge \ i_0(t) \geq 0 \ )$
Resistor0	:	$(\ \mathcal{V}_m \ \mid \ v_0(t) - v_1(t) = i_0(t) R_0 \ )$
Resistor1	:	$(\ \mathcal{V}_m \ \mid \ v_2(t) = i_2(t) R_1 \ )$
Source	:	$(\ \mathcal{V}_m \ \mid \ v_0'(t) = \sin(2\pi f \, clock(t)) \ )$

Figure 7.6: Half wave rectifier voltages

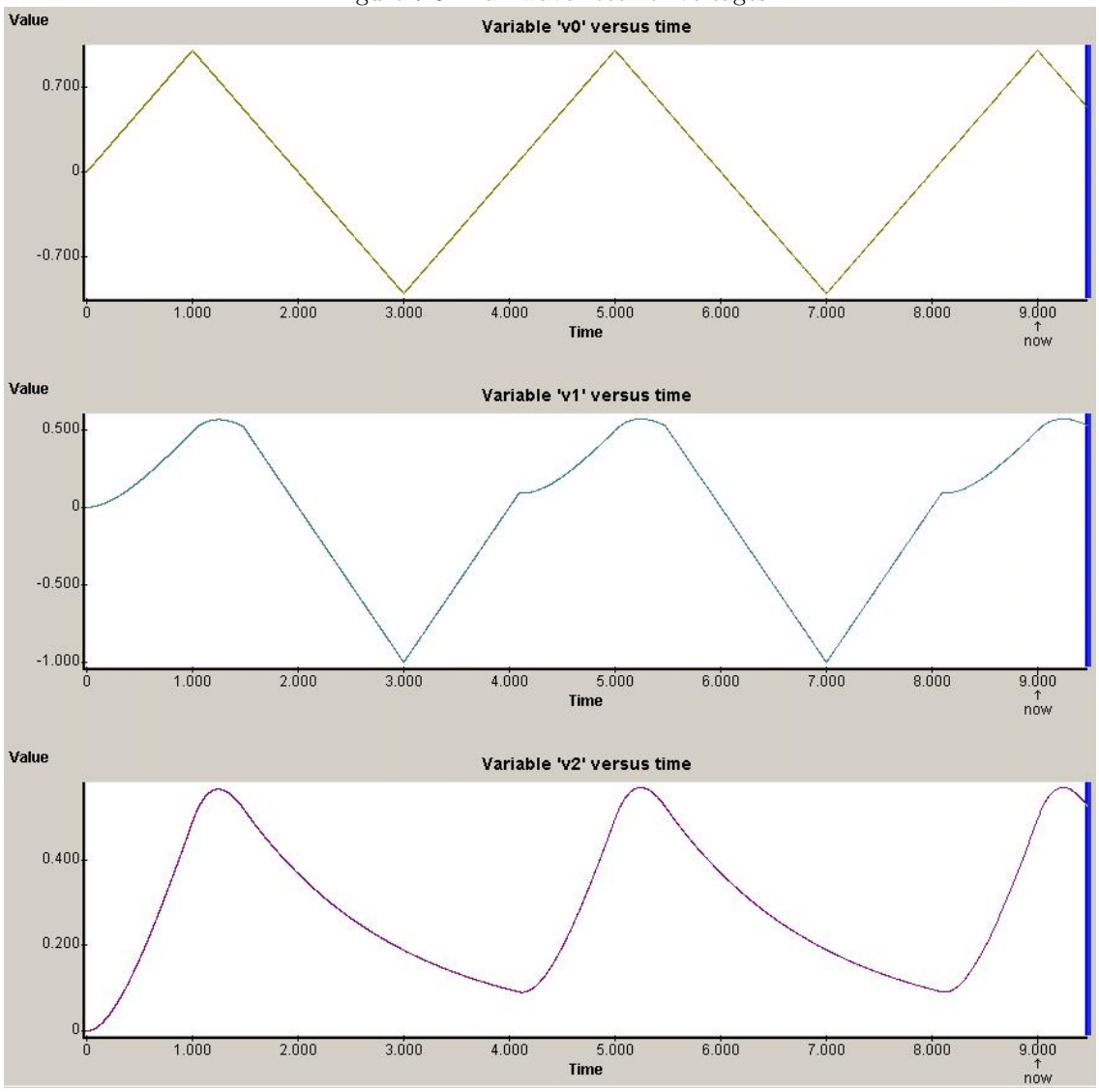


Figure 7.7: Half wave rectifier currents

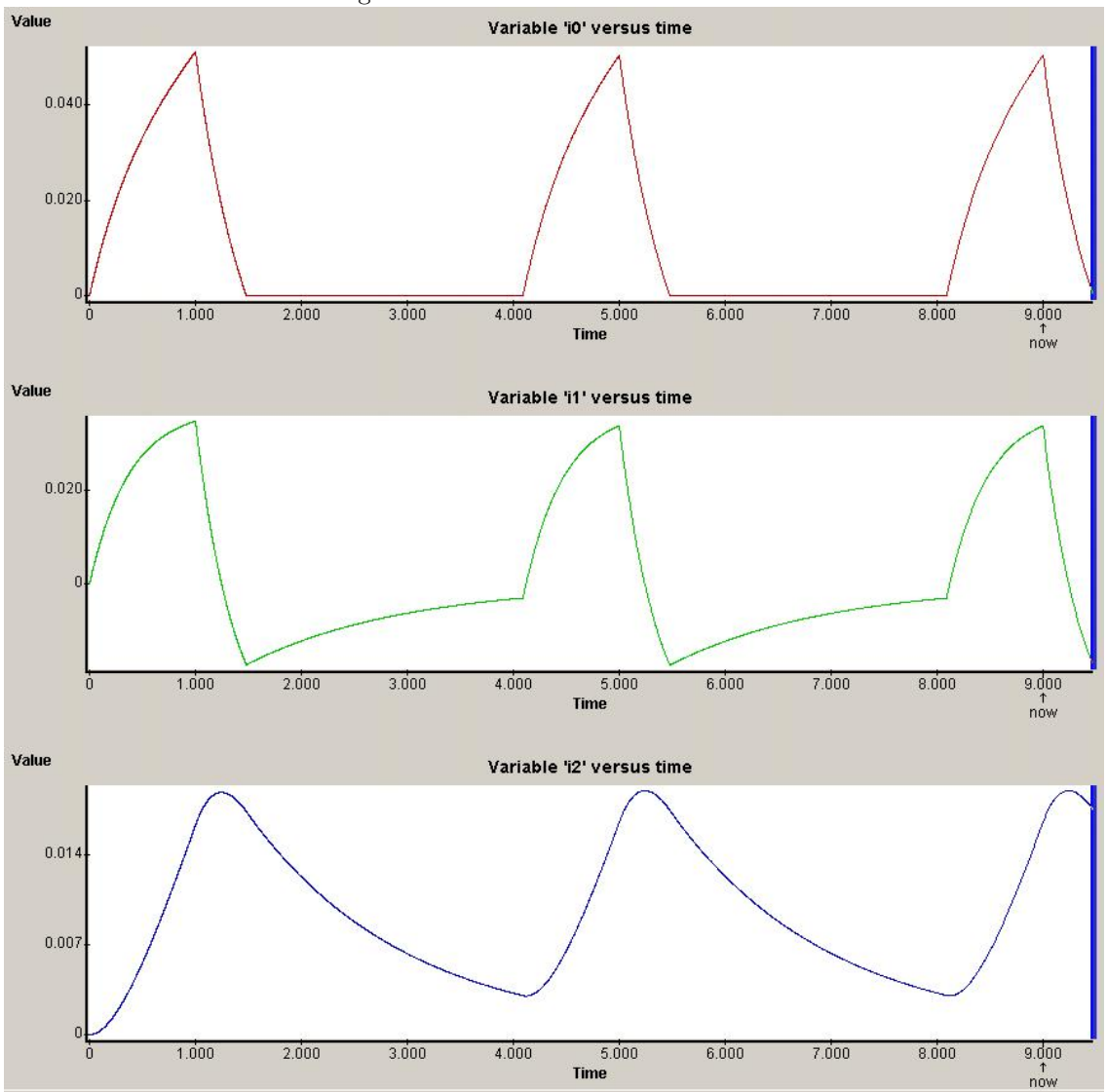


Table 7.7:

$\mathcal{A} = \{\}$

$\mathcal{V}_{\mathrm{m}} = \{\mathrm{clock},\ \mathrm{i0},\ \mathrm{i1},\ \mathrm{i2},\ \mathrm{v0},\ \mathrm{v1},\ \mathrm{v2}\}$

$f = 10;\ R0 = 10;\ R1 = 30;\ C0 = 50 \times 10^{-5};\ e = 0.001;$

Rectifier	:	SystemOn $\oplus$ SystemOff
SystemOff	:	(System $\parallel$ DiodeOff) $\rhd$ Switch
SystemOn	:	(System $\parallel$ DiodeOn) $\rhd$ Switch
System	:	Clock $\parallel$ Source $\parallel$ Resistor0 $\parallel$ Capacitor0 $\parallel$ Resistor1
Capacitor0	:	$\left(\ \mathcal{V}_{\mathrm{m}}\ \middle|\ C_0 v2'(t) = i_1(t)\ \right)$
Clock	:	$\left(\ \mathrm{clock}\ \middle|\ clock'(t) = 1\ \right)$
DiodeOff	:	$\left(\ \mathcal{V}_{\mathrm{m}}\ \middle|\ i_0(t) = 0 \wedge i_1(t) = -i_2(t) \wedge v_1 \leq v_2 + e\ \right)$
DiodeOn	:	$\left(\ \mathcal{V}_{\mathrm{m}}\ \middle|\ v_1(t) = v_2(t) \wedge i_0(t) = i_1(t) + i_2(t) \wedge i_0 \geq 0 - e\ \right)$
Resistor0	:	$\left(\ \mathcal{V}_{\mathrm{m}}\ \middle|\ v_0(t) - v_1(t) = i_0(t) R_0\ \right)$
Resistor1	:	$\left(\ \mathcal{V}_{\mathrm{m}}\ \middle|\ v_2(t) = i_2(t) * R_1\ \right)$
Source	:	$\left(\ \mathcal{V}_{\mathrm{m}}\ \middle|\ v_0'(t) = \sin(2\pi f\, clock(t))\ \right)$
Switch	:	$\left[\ i_0,\ i_1,\ i_2,\ v_2\ \middle|\ i_0^- \leq 0 + e \wedge v_1^- \geq v_2^- - e \wedge i_0^+ = 0 \wedge i_1^+ = 0 \wedge i_2^+ = 0 \wedge v_2^+ = v_1^-\ \right]$ $\gg$ SystemOn $\oplus$ $\left[\ i_0,\ i_1,\ i_2,\ v_2\ \middle|\ i_0^- \leq 0 + e \wedge v_1^- \leq v_2^- + e \wedge i_0^+ = 0 \wedge i_1^+ = 0 \wedge i_2^+ = 0 \wedge v_2^+ = v_1^-\ \right]$ $\gg$ SystemOff

Chapter 8

Conclusions

The goal of this thesis was to build a simulator for HyPA. It should be capable of simulating a number of classes of HyPA models. At least one clause formalism should be implemented that supports certain classes of differential equations in the predicates. The simulator was to visualize process behavior, i.e. the actions and the model variable jumps and flows. Two modes of simulation were required: step-wise and random.

To ease simulation of models that have been created for the HyPA linearization tool, the input file format of that tool should be read by the simulator. Also, the simulator should be able to cope with model variables for which behavior is not completely defined.

The simulator was to be modular with respect to file formats and clause formalisms.

A simulator has been built, and it fulfills most requirements. A simple but adequate development process was used to gather requirements, design and implement the simulator. Case studies were performed several times during the process to gather more requirements and to test the simulator.

Classes of HyPA models

The simulator repeatedly calculates the first possible transitions of a HyPA model. To do so, we defined an inductive function on the structure of the model. Showing soundness of the function (i.e. that it only produces transitions that the model can perform) was easy, since the function is based on the HyPA operational semantics. Completeness and termination were proven for the class of *substitutably guarded* HyPA models. The required $\mathrm{HyPA}_{\mathrm{lin}}$ and $\mathrm{HyPA}^{\mathrm{nt}}$ classes of models are contained in the class of substitutably guarded models.

The simulator exceeds the requirements by allowing all possible HyPA models to be simulated. It does not guarantee completeness for models outside the substitutably guarded class. An artificial termination criterium for the calculation is introduced, through which the user can indicate how much effort to spend to find first transitions.

Extensions to HyPA

HyPA models can become cumbersome to write, in particular when many sub-processes are similar. To solve this, new HyPA operators have been defined, and the simulator allows for some syntactic sugar in its input format. The new operators are action renaming and variable renaming. They are defined through operational semantics. Robust bisimulation is proven to be a congruence for them, so the HyPA derivation system is still correct for HyPA models that use them. The syntactic sugar comprises easier declaration of actions and communications, notation for the entire set of model variables, and notation for all the 'halves' of the communications.

Model variable abstraction is implemented only partially, and this is similar to the state of affairs in the linearization tool.

Clause formalisms

The simulator implements two clause formalisms, both related to Mathematica. One solves the clauses numerically, and the other symbolically. A generic clause formalism that allows for any solver to be used, turned out to be cumbersome to implement and to be restrictive to the user. The mathematica-related formalisms allow the use of most Mathematica constructs that are available. They fulfill all requirements about classes of (differential) equations to be handled.

Numeric calculation does have its limitations: forced ending of a flow is not always calculated correctly, and the precision and accuracy of the calculation is not really controllable. The latter is a property of Mathematica. Because of this, models often need to be rewritten to let them behave as desired under numeric calculation.

Symbolic calculation can be used for a more limited class of differential equations than in the numeric case, but it works as expected. Models written for analysis purposes do not have to be rewritten.

c_{jmp} constructs turn out to be difficult to solve. This is mainly a problem of Mathematica. Therefore, the currently implemented clause formalisms do not implement c_{jmp} solving. The simulator do‘es allow for HyPA models with c_{jmp} to be read, and the design is such that c_{jmp}-solving formalisms can easily be added. The first transitions function introduces c_{jmp} constructs for certain HyPA models. The simulator rewrites models such that that does not occur. The rewrite process is proven to produce robustly bisimilar models. This could not be achieved by using the HyPA derivation system, but had to be done using witnessing relations. The HyPA derivation system turns out not to handle the case that the jump at the beginning of a flow clause is taken over by a reinitialization.

Cases

The example models show that the implemented visualizations are indeed useful. Also, the additional HyPA operators turn out to be useful. An additional renaming operator for constants in clause predicates could be useful as well.

Step-wise and random simulation work as expected. After some optimizations, first transitions are calculated quickly. The renaming-operators help in optimizing the first transitions calculation. Some models produce very many first transitions, only some of which are actually solvable. Such models take long to simulate, because each first transition has to be looked at with the help of Mathematica.

For models with underspecific clauses (i.e. where model variable behavior is not completely specified), the simulator asks for more clause predicates during simulation. For random simulation, it is therefore useful to rewrite such models in a more specific form.

Architecture

The design of the simulator is modular in the requested respects: clause formalisms and input file formats. In addition, it is modular with respect to new HyPA operators; the renaming operators were easy to add. The architecture is such that other user interfaces and other engines can be used. This has proven useful in the design of the user interface.

Further research

There is room for further work in several respects: unfulfilled requirements, completeness for other classes of models, and useful additions to the simulator. Unfulfilled requirements include:

- Implementing a generic clause formalism to which different solvers can be connected (but this is restrictive to the user);
- A different way of indicating steps to undo;

- Having a predicate as a stop criterium for random simulation;
- For other input file formats to be read, there must be some selection mechanism for input formats in the user interface;
- Simulation of models with abstraction anywhere;
- Filling in random behavior for unspecified behavior.

In order to prove completeness of the first transitions function, one of the two failed approaches to the function can be explored further. The protectedness-property (section 4.1.2.1) can be investigated further to match bounded non-determinism to a structural property of a model. Also, another way of proving the fixpoint approach (section 4.1.2.2) could be tried.

Additions to the simulator could include:

- Providing more information to the user on underspecific clauses;
- More error-checking of clause predicates;
- Having a new operator to rename constants within a clause predicate;
- Having simulator settings for a model in a file per model.

Bibliography

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. charon, a language for modular specification of hybrid systems. Technical report, University of Pennsylvania, Department of Computer and Information Science, 2000. URL `http://www.cis.upenn.edu/mobies/charon/`.

[2] J.C.M. Baeten and W.P.Weijland. *Process Algebra*. Press Syndicate of the University of Cambridge, 1990, ISBN: 0521400430.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999, ISBN: 0201616416.

[4] Roel Bloo, Herman Geuvers, and Jozef Hooman. *Syllabus of the course Computational Models (2R070)*. Technische Universiteit Eindhoven, 2000.

[5] Ed Brinksma, Tomas Krilavicius, and Yaroslav S. Usenko. Behavioural hybrid process calculus (draft). Technical report, University of Twente, 2005.

[6] P.J.L Cuijpers. *Hybrid Process Algebra*. PhD thesis, Technische Universiteit Eindhoven, 2004.

[7] P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191–245, February 1, 2005.

[8] Erich Gamma, Richard Helm, and Ralph Johnson. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995, ISBN: 0201633612.

[9] Hyvisual. URL `http://www.ptolemy.eecs.berkely.edu/ptolemyII/hyvisual2.2/index.htm`.

[10] Erwin Kreyszig. *Advanced Engineering Mathematics*. John Wiley and Sons, 6th edition edition, 1998, ISBN: 0 471 85824 2.

[11] Mohammad Reza Mousavi, Michel Reniers, and Jan Friso Groote. Congruence for SOS with data. In *Proceedings of Nineteenth Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 302–313, Turku, Finland, 2004. IEEE Computer Society Press.

[12] G.D. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[13] Ptolemy plot. URL `http://www.ptolemy.eecs.berkely.edu/java/ptplot/index.htm`.

[14] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Formal semantics of hybrid chi. In Kim Guldstrand Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems: FORMATS 2003*, volume 2791 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2004.

[15] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[16] Y.S. Usenko. *Linearization in μCRL*. PhD thesis, Technische Universiteit Eindhoven, 2002.

[17] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. Technical Report Computing Science Report 04-37, Eindhoven University of Technology, Department of Mathematics and Computing Science, 2004.

[18] P.C.W. van den Brand. Linearization of hybrid processes. Master's thesis, Technische Universiteit Eindhoven, 2004.

[19] P.C.W. van den Brand, M.A. Reniers, and P.J.L. Cuijpers. Linearization of hybrid processes. *Journal of Logic and Algebraic Programming*, 2005.

[20] Huub van den Broek. Practical assignment: Visualizing hybrid systems, Technische Universiteit Eindhoven, 2002.

Appendix A

Mathematical definitions

The following definitons were taken from [10].

Definition A.1 (Differential equation)
A *differential equation* is an equation that contains a derivative of an unknown function which we want to determine from the equation.

Definition A.2 (Ordinary differential equation (ODE))
A *ordinary differential equation* (ODE) is an equation that involves one or several derivatives of an unspecified function y of x. The equation may also involve y itself, given functions of x, and constants. [...] The word *ordinary* distinguishes such an equation from a *partial differential equation* which involves partial derivatives of an unspecified function of two or more independent variables.

Examples of ODEs are:

$$y' = \cos(x)$$
$$y'' + 4y = 0$$
$$x^2 y''' y' + 2e^x y'' = (x^2 + 2)y^2$$

An example of a partial differential equation is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Definition A.3 (Order of a differential equation)
We say that a differential equation is of *order* n if the nth derivative of y with respect to x is the highest derivative in the equation.

Definition A.4 (First-order differential equation)
A *first-order differential equation* contains only y', and may contain y and given functions of x.

Many times we are only interested in a particular solution for a differential equation that satisfies initial conditions:

Definition A.5 (Initial condition)
Let x_0 and y_0 be constants.
An *initial condition* is a predicate of the form $y(x_0) = y_0$.

Definition A.6 (Initial value problem)
An *initial value problem* is a first-order differential equation together with an initial condition.

Definition A.7 (First-order linear differential equation)
A differential equation is called *linear* if it can be written $y' + p(x)y = r(x)$, where p and r may be any given functions of x. Function p is called a *coefficient* of the equation.

Linear differential equations have a known general solution $y(x) = e^{-h}(\int e^h r dx + c)$ where $h = \int p(x)dx$.

Appendix B

Input restrictions

Here, different restrictions on HyPA specifications are described. These restrictions are used in the requirements (chapter 3) to specify which classes of HyPA models need to be simulated.

Linear Recursive Specification

From [18]:

Definition B.1 (Linear Recursive Specification)
A *linear recursive specification* $\langle\, t \,|\, E \,\rangle$ is a recursive specification that satisfies the following requirements:

1. the initial term t is either a recursion variable in E (i.e. X) or the variable abstraction of a reinitialization on such a variable (i.e. $|[\ V \mid d \gg X\ ]|$), and
2. the right-hand sides of all recursive equations in E are linear terms.

A term is linear, if it has the BNF form p:

$$p ::= \delta \quad | \quad d \gg a \quad | \quad d \gg a \odot X \quad | \quad d \gg c \rhd X \quad | \quad p \oplus p$$

$\mathrm{HyPA}_{\mathrm{lin}}$ input restriction

Here is the definition of a $\mathrm{HyPA}_{\mathrm{lin}}$ specification from [18]:

Definition B.2 ($\mathrm{HyPA}_{\mathrm{lin}}$ Specification)
A *$HyPA_{\mathrm{lin}}$ specification* is a recursive specification $\langle\, t \,|\, E \,\rangle$ that satisfies the following three restrictions:

1. $\langle\, t \,|\, E \,\rangle$ is *substitutably guarded* (definition 2.12);
2. t is in $\mathrm{HyPA}_{\mathrm{par}}$ form, and
3. the right-hand sides of all recursive equations in E are in $\mathrm{HyPA}_{\mathrm{lin}}$ form.

Definition B.3 ($\mathrm{HyPA}_{\mathrm{par}}$ Form)
The $\mathrm{HyPA}_{\mathrm{par}}$ form is defined as the form p:

$$\begin{array}{lll} p & ::= & |[\ V \mid q\ ]| \quad | \quad q \\ q & ::= & X \quad | \quad q \parallel q \quad | \quad \partial_H(q) \end{array}$$

Definition B.4 ($\mathbf{HyPA_{lin}}$ Form)
The $\mathrm{HyPA_{lin}}$ form is defined as follows:

$$p \quad ::= \quad a \mid X \mid \delta \mid c \mid p \oplus p \mid p \odot p \mid d \gg p \mid c \blacktriangleright p \mid c \rhd p$$

$\mathbf{HyPA^{nt}}$ input restriction

A $\mathrm{HyPA^{nt}}$ specification differs from a $\mathrm{HyPA_{lin}}$ specification in that abstraction is no longer allowed, and that parallel processes cannot terminate.

Definition B.5 ($\mathbf{HyPA^{nt}}$ Specification)
A *HyPA*$^{\mathrm{nt}}$ *specification* is a recursive specification $\langle t \mid E \rangle$ that satisfies the following three restrictions:

1. $\langle t \mid E \rangle$ is *substitutably guarded*;
2. t is in $\mathrm{HyPA^{nt}_{par}}$ form, and
3. the right-hand sides of all recursive equations in E are in $\mathrm{HyPA^{nt}_{lin}}$ form.

Definition B.6 ($\mathbf{HyPA^{nt}_{par}}$ Form)
The $\mathrm{HyPA^{nt}_{par}}$ form is defined as the form q:

$$q \quad ::= \quad X \mid q \parallel q \mid \partial_H(q)$$

Definition B.7 ($\mathbf{HyPA^{nt}_{lin}}$ Form)
The $\mathrm{HyPA^{nt}_{lin}}$ form is defined as the form p:

$$\begin{array}{lll} p & ::= & X \mid \delta \mid c \mid p \oplus p \mid q \odot X \mid d \gg p \mid c \blacktriangleright p \mid c \rhd p \\ q & ::= & a \mid X \mid \delta \mid c \mid q \oplus q \mid q \odot q \mid d \gg q \mid q \blacktriangleright q \mid q \rhd q \end{array}$$

Appendix C

Proofs for $F_E(t)$

Here, the first transitions function $F_E(t)$ defined in section 4.1.1 is proven correct. First we prove that the induction is well-defined, and then that it actually gives the first transitions.

C.1 $F_E(t)$ is well-defined for substitutably guarded models

Definition C.1 (Weight function)
For a substitutably guarded specification $\langle\, t \mid E \,\rangle$, define the *weight of* $\langle\, t \mid E \,\rangle$, notation $w(\langle t|E\rangle)$ as a pair $\langle\, s, o \,\rangle$, where:

- s is the number of substitutions needed to make t completely guarded, and
- o is the number of operators in t

Since t is finite and $\langle\, t \mid E \,\rangle$ is guarded, $w(\langle t|E\rangle)$ is finite.

Theorem C.1 (Well-definedness)
$F_E(t)$ is well-defined for substututably guarded specifications.

Proof
The weight of specifications in the right-hand side of all rules in the definition for $F_E(t)$ decreases with respect to the lexicographic order.

Rules 1 through 4: $w(\langle t|E\rangle) = \langle\, 0, 0 \,\rangle$ and F_E does not occur in the right-hand-side.

Rules 5, 6, 8, 10 through 13: the number of operators in p and q is one less than in t, and no guarded recursion variables become unguarded, so if $w(\langle t|E\rangle) = \langle\, s, o \,\rangle$ then $w(\langle p|E\rangle) = w(\langle q|E\rangle) = \langle\, s, o - 1 \,\rangle$.

Rule 7: the number of operators in p and q is one less than in t. However, possibly a guarded recursion variable becomes unguarded. This can only be the case when $p = a$ for some action a. This implies however that $F_E(p) = \{\langle\, [true] \gg a, \epsilon \,\rangle\}$, and thus $F_E(q)$ is never evaluated.

Rule 9: Only $F_E(p)$ is evaluated in $F_E(p \rhd q)$ which decreases the number of operators as well as the number of (possibly unguarded) recursion variables. Therefore we have $w(\langle p|E\rangle) < w(\langle p \rhd q|E\rangle)$.

Rule 14: This is a substitution of X by $rhs(X)$ so if $w(\langle X|E\rangle) = \langle\, s, o \,\rangle$ then $w(\langle rhs(X)|E\rangle) = \langle\, s - 1, o' \,\rangle$ for some o'.

Furthermore, the precondition for $F_E(t)$ is met in recursive applications: $\langle t'|E\rangle$ in recursive applications $F_E(t')$ is still substitutably guarded, since substitutable guardedness only depends on E.

Theorem C.2 (Form of $F_E(t)$)
By construction, the value returned by $F_E(t)$ is a set with elements of the form

$$\langle\, d \gg a, p \,\rangle \quad | \quad \langle\, d \gg c, p \,\rangle \quad | \quad \langle\, d \gg \epsilon, \checkmark \,\rangle$$

Here, $\langle p|E\rangle$ is substitutably guarded, since substitutable guardedness only depends on E.

C.2 $F_E(t)$ is sound

We prove that every transition that is in $F_E(t)$ can be performed by the model.

Theorem C.3 (Soundness: termination)
Termination in $F_E(t)$ implies that $\langle\, t \,|\, E \,\rangle$ can terminate. To prove:

$$\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark) \Rightarrow \langle\, t, v \,\rangle\checkmark$$

Proof
Proof by induction on $w(\langle t|E\rangle)$. We suppose that $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$ and prove for each case that $\langle\, t, v \,\rangle\checkmark$.

Case $w(\langle t|E\rangle) = 0$: t has no operators and no unguarded occurrences of recursion variables, so $t ::= \epsilon \quad | \quad a \quad | \quad c \quad | \quad \delta$. Thus, one of rules 1 through 4 of F_E has been applied to t and trivially the theorem holds.

Case $w(\langle t|E\rangle) > 0$: We use case distinction on the rule of F_E used first. Rules 1 through 4 cannot have been used.

Rule 5: $t = d' \gg p$, $d = d' \sim e$, $\langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(p)$ for some d', e', p. From $\langle\, d \gg \epsilon, v \,\rangle\checkmark$ we conclude $\exists_{v',v''}((v, v') \models d'$ and $(v', v'') \models e$ and $\langle\, \epsilon, v'' \,\rangle\checkmark)$. OS-rule 4 gives $\langle\, e \gg \epsilon, v' \,\rangle\checkmark$. Induction $\langle\, p, v' \,\rangle\checkmark$. OS-rule 4 gives $\langle\, t, v \,\rangle\checkmark$.

Rule 6: $t = p \oplus q$, $\langle\, d \gg x, t' \,\rangle \in F_E(p)$ or $\langle\, d \gg x, t' \,\rangle \in F_E(q)$ for some p, q. By induction and OS-rule 6.

Rule 7: $t = p \odot q$, $d = d'^{?} \wedge e^{?}$, $\langle\, d' \gg \epsilon, \checkmark \,\rangle \in F_E(p)$, $\langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)$. OS-rule 4 and the definition of $^{?}$ gives $\exists_{v',v'''}((v, v') \models d' \wedge (v, v''') \models e)$. By OS-rules 4 and 1 we get $\langle\, d' \gg \epsilon, v \,\rangle\checkmark$ and $\langle\, e \gg \epsilon, v \,\rangle\checkmark$. Induction: $\langle\, p, v \,\rangle\checkmark$ and $\langle\, q, v \,\rangle\checkmark$. OS-rule 8 gives $\langle\, t, v \,\rangle\checkmark$.

Rule 8: $t = p \blacktriangleright q, \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)$ for some p, q. By induction and OS-rule 11.

Rule 9: $t = p \triangleright q, \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)$ for some p, q. By induction and OS-rule 11.

Rule 10: $t = p \parallel q$, $d = d'^{?} \wedge e^{?}$, $\langle\, d' \gg \epsilon, \checkmark \,\rangle \in F_E(p)$, $\langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)$. OS-rule 4 and the definition of $^{?}$ gives $\exists_{v',v'''}((v, v') \models d' \wedge (v, v''') \models e)$. By OS-rules 4 and 1 we get $\langle\, d' \gg \epsilon, v \,\rangle\checkmark$ and $\langle\, e \gg \epsilon, v \,\rangle\checkmark$. Induction: $\langle\, p, v \,\rangle\checkmark$ and $\langle\, q, v \,\rangle\checkmark$. OS-rule 15 gives $\langle\, t, v \,\rangle\checkmark$.

Rule 11: $t = p \mathbin{\lfloor\!\lfloor} q$ for some p, q. By construction the hypothesis does not hold.

Rule 12: $t = p \mid q$ for some p, q. Analogous to Rule 10.

Rule 13: $t = \partial_H(p)$. By induction and OS-rule 22.

Rule 14: $t = X$. By induction and OS-rule 23.

Theorem C.4 (Soundness: transitions)
All transitions in $F_E(t)$ can be performed by $\langle\, t \,|\, E \,\rangle$. To prove:

$$\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle) \Rightarrow \langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$$

Proof
Proof by induction on $w(\langle t|E\rangle)$. We suppose $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$. For each case below, we prove that $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$

Case $w(\langle t|E\rangle) = 0$: t has no operators and no unguarded occurrences of recursion variables, so $t ::= \epsilon \mid a \mid c \mid \delta$. Thus, one of rules 1 through 4 of F_E has been applied to t and trivially the theorem holds.

Case $w(\langle t|E\rangle) > 0$: We use case distinction on the rule of F_E used first. Rules 1 through 4 cannot have been used.

Rule 5: $t = d' \gg p,\ d \gg x = (d' \sim e) \gg p'',\ \langle\, e \gg p'', p' \,\rangle \in F_E(p),\ t' = p'$ for some d', e, p, p', p''. By OS-rule 5 $\exists_{v',v'''}((v', v''') \models d'$ and $(v''', v') \models e$ and $\langle\, p'', v' \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle$. OS-rule 5 read in the other direction now gives us $\langle\, e \gg p'', v''' \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle$. Induction $\langle\, p, v''' \,\rangle \rightarrow^l \langle\, p', v'' \,\rangle$ and OS-5 gives $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.

Rule 6: $t = p \oplus q,\ \langle\, d \gg x, t' \,\rangle \in F_E(p)$ or $\langle\, d \gg x, t' \,\rangle \in F_E(q)$. Induction: $\langle\, p, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$ or $\langle\, q, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$. OS-rule 7 gives $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.

Rule 7: $t = p \odot q$ for some p, q. We have two cases.

- Case 1: $t' = p' \odot q,\ \langle\, d \gg x, p' \,\rangle \in F_E(p)$ for some p'. Induction: $\langle\, p, v \,\rangle \rightarrow^l \langle\, p', v'' \,\rangle$ so OS-rule 9 gives $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.
- Case 2: $t' = q',\ d \gg x = (d'^? \sim e) \gg q'',\ \langle\, e \gg q'', q' \,\rangle \in F_E(q),\ \langle\, d' \gg \epsilon, \checkmark \,\rangle \in F_E(p)$ for some d', e', q', q''. By OS-rule 5 we have $\exists_{v''''}((v, v) \models d'^? \wedge (v, v''') \models e \wedge \langle\, q'', v''' \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.
  - From $(v, v) \models d'^?$ we conclude $\exists_{v'}((v, v') \models d')$. OS-rule 4 gives $\langle\, d' \gg \epsilon, v \,\rangle\checkmark$. Theorem 4.3 now gives $\langle\, p, v \,\rangle\checkmark$.
  - OS-rule 5 gives us $\langle\, e \gg q'', v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle$. Induction $\langle\, q, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle$.

  From $\langle\, p, v \,\rangle\checkmark$ and $\langle\, q, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle$ we get by OS-rule 10 that $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.

Rule 8: $t = p \blacktriangleright q$ for some p, q. We have two cases.

- Case 1: $t' = p' \blacktriangleright q,\ \langle\, d \gg x, p' \,\rangle \in F_E(p)$ for some p'. Induction: $\langle\, p, v \,\rangle \rightarrow^l \langle\, p', v'' \,\rangle$. By OS-rule 12 $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.
- Case 2: $\langle\, d \gg x, t' \,\rangle \in F_E(q)$. By induction and OS-rule 14 $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.

Rule 9: $t = p \rhd q,\ t' = p' \blacktriangleright q,\ \langle\, d \gg x, p' \,\rangle \in F_E(p)$ for some p'. Induction: $\langle\, p, v \,\rangle \rightarrow^l \langle\, p', v'' \,\rangle$. By OS-rule 12 $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.

Rule 10: $t = p \parallel q$ for some p, q. We have six cases.

- Case 1: $t' = p' \parallel q',\ d \gg x = ((d_1 \sim c_{1jmp}) \wedge (d_2 \sim c_{2jmp})) \gg c_1 \wedge c_2,\ \langle\, d_1 \gg c_1 \,\rangle \in F_E(p),\ \langle\, d_2 \gg c_2 \,\rangle \in F_E(q),\ l = \sigma$ for some $p', q', d_1, d_2, c_1, c_2, \sigma$. From OS-rules 3, 5 we know $\exists_{v',v_3,v_4,t''}((v, v_3) \models d_1 \wedge (v_3, v') \models c_{1jmp} \wedge (v, v_4) \models d_2 \wedge (v_4, v') \models c_{2jmp} \wedge (v', \sigma) \models c_1 \wedge (v', \sigma) \models c_2 \wedge dom(\sigma) = [0..t''] \wedge v'' = \sigma(t'') \wedge s' = c_1 \wedge c_2$. The definition of c_{jmp} implies

that $(v_3, \sigma) \models c_1 \wedge (v_4, \sigma) \models c_2$. Now by OS-rules 3, 5 $\langle\, d_1 \gg c_1, v \,\rangle \rightsquigarrow^{\sigma} \langle\, c_1, \sigma(t'') \,\rangle$ and $\langle\, d_2 \gg c_2, v \,\rangle \rightsquigarrow^{\sigma} \langle\, c_2, \sigma(t'') \,\rangle$. Induction: $\langle\, p, v \,\rangle \rightsquigarrow^{\sigma} \langle\, p', v'' \,\rangle$ and $\langle\, q, v \,\rangle \rightsquigarrow^{\sigma} \langle\, q', v'' \,\rangle$. By OS-rule 16 we now get $\langle\, t, v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$.

- Case 2: $t' = p',\ d \gg x = e^{?} \sim d' \gg c,\ \langle\, d' \gg c, p' \,\rangle \in F_E(p),\ \langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q)$ for some p', d', e, c. By OS-rule 5 $\exists_{v'}((v, v) \models e^{?} \wedge (v, v') \models d' \wedge \langle\, c, v' \,\rangle \rightarrow^{l} \langle\, c, v'' \,\rangle$. Thus $\exists_{v'''}((v, v''') \models e)$. Theorem 4.3 gives $\langle\, q, v \,\rangle\checkmark$. OS-rule 5 and induction gives $\langle\, p, v \,\rangle \rightarrow^{l} \langle\, p', v' \,\rangle$ and OS-rule 17 gives $\langle\, t, v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$.

- Case 3: analogous to case 2.

- Case 4: $t' = p' \,\|\, q,\ x = a,\ \langle\, d \gg a, p' \,\rangle \in F_E(p)$ for some p', a. OS-rules 5 and 2 yield $l = (a, v')$ for some v'. Induction: $\langle\, p, v \,\rangle \rightarrow^{l} \langle\, p', v'' \,\rangle$. OS-rule 18 gives $\langle\, t, v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$.

- Case 5: analogous to case 4.

- Case 6: $t' = p' \,\|\, q',\ d \gg x = d_1 \wedge d_2 \gg a,\ \langle\, d_1 \gg a_1, p' \,\rangle \in F_E(p),\ \langle\, d_1 \gg a_1, q' \,\rangle \in F_E(q),\ \gamma(a_1, a_2) = a$ for some $p', q', d_1, d_2, a, a_1, a_2$. By OS-rules 2 and 5 $(v, v'') \models d_1 \wedge (v, v'') \models d_2)$. Thus, by OS-rule 2 and 5 $\langle\, d_1 \gg a_1 \,\rangle \rightarrow^{(a_1, v'')} \langle\, \epsilon, v'' \,\rangle$ and $\langle\, d_2 \gg a_2 \,\rangle \rightarrow^{(a_2, v'')} \langle\, \epsilon, v'' \,\rangle$. Induction: $\langle\, p, v \,\rangle \rightarrow^{(a_1, v'')} \langle\, p', v'' \,\rangle$ and $\langle\, q, v \,\rangle \rightarrow^{(a_2, v'')} \langle\, q', v'' \,\rangle$. Now OS-rule 19 gives $\langle\, t, v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$.

Rule 11: $t = p \between q$ for some p, q. This proof is similar to Rule 10, case 4.

Rule 12: $t = p \,|\, q$ for some p, q. This proof is similar to Rule 10, cases 1, 2, 3, and 6.

Rule 13: $t = \partial_H(p)$ for some H, p. We have two cases:

- Case 1: $t' = \partial_H(p'),\ x = a,\ \langle\, d \gg a, p' \,\rangle \in F_E(p),\ a \notin H$ for some p', a. Induction and OS-rules 2 and 5 give $\langle\, p, v \,\rangle \rightarrow^{l} \langle\, p', v'' \,\rangle$ and $l = (a, v'')$. OS-rule 20 gives $\langle\, t, v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$.

- Case 2: $t' = \partial_H(p'),\ x = c,\ \langle\, d \gg c, p' \,\rangle \in F_E(p)$ for some p', c. Induction and OS-rules 3 and 5 give $\langle\, p, v \,\rangle \rightarrow^{l} \langle\, p', v'' \,\rangle$ and $l = \sigma$ for some σ. OS-rule 21 gives $\langle\, t, v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$.

Rule 14: $t = X,\ \langle\, d \gg x, t' \,\rangle \in F_E(rhs(X))$. By induction $\langle\, rhs(X), v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$. By OS-rule 24 we get $\langle\, t, v \,\rangle \rightarrow^{l} \langle\, t', v'' \,\rangle$.

C.3 $F_E(t)$ is complete for substitutably guarded models

We prove that everything that a model can do, is in $F_E(t)$.

Theorem C.5 (Completeness: termination)
Termination of $\langle\, t \,|\, E \,\rangle$ is in $F_E(t)$. To prove:

$$\langle\, t, v \,\rangle\checkmark \Rightarrow \exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$$

Proof
Proof by induction on the length of the derivation of $\langle\, t, v \,\rangle\checkmark$. We make no distinction between base and step cases. We use case distinction on the OS-rule applied in the last step in the derivation. This can only be one of OS-rules 1, 4, 6, 8, 11, 13, 15, 22, 23.

OS-rule 1: $t = \epsilon$ and the definition of $F_E(\epsilon)$ combined with axiom 9-1 implies $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$.

OS-rule 4: $t = d \gg p$ for some d, p and $\exists_{v'}((v, v') \models d$ and $\langle\, p, v' \,\rangle\checkmark)$. By induction $\exists_e(\langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, e \gg \epsilon, v' \,\rangle\checkmark)$. By OS-rule 4 this implies that $\exists_{v''}((v', v'') \models e$ and $\langle\, \epsilon, v'' \,\rangle\checkmark)$. Therefore $(v, v'') \models d \sim e$. By OS-rule 1, 4 we have $\langle\, d \sim e \gg \epsilon, v \,\rangle\checkmark$. By definition of $F_E(d \gg p)$ we now have $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$.

OS-rule 6: We have for some p, q that $t = p \oplus q$ and either $\langle\, p, v \,\rangle\checkmark$ or $\langle\, q, v \,\rangle\checkmark$. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$ or $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(q) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$. Thus, by definition of $F_E(p \oplus q)$ we have $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$.

OS-rule 8: $t = p \odot q, \langle\, p, v \,\rangle\checkmark, \langle\, q, v \,\rangle\checkmark$ for some p, q. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$ and $\exists_e(\langle\, e \gg \epsilon, \checkmark \,\rangle \in F_E(q) \wedge \langle\, e \gg \epsilon, v \,\rangle\checkmark)$. By definition of $F_E(p \odot q)$ we have $\langle\, (d^? \wedge e^?) \gg \epsilon, \checkmark \,\rangle \in F_E(t)$. From $\langle\, d \gg \epsilon, v \,\rangle\checkmark$ and $\langle\, e \gg \epsilon, v \,\rangle\checkmark)$ we know $\exists_{v', v''}((v, v') \models d \wedge (v, v'') \models e)$. Therefore, $(v, v) \models d^? \wedge e^?$ and thus $\langle\, (d^? \wedge e^?) \gg \epsilon, v \,\rangle\checkmark$.

OS-rule 11: For some p, p', q, $\langle\, p, v \,\rangle\checkmark$ and $t = p \blacktriangleright q$ or $t = p \rhd q$. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$. By definition of $F_E(p \blacktriangleright q)$ and $F_E(p \rhd q)$ we have $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$.

OS-rule 13: For some p, q we have $t = p \blacktriangleright q$ and $\langle\, q, v \,\rangle\checkmark$. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(q) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$. Thus by the definition of $F_E(p \blacktriangleright q)$ we have $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$.

OS-rule 15: For some p, q we have $\langle\, p, v \,\rangle\checkmark$ and $\langle\, q, v \,\rangle\checkmark$ and either $t = p \parallel q$ or $t = p \mid q$. This proof is analogous to the one for OS-rule 8.

OS-rule 22: For some p, we have $t = \partial_H(p)$ and $\langle\, p, v \,\rangle\checkmark$. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$. By definition of $F_E(\partial_H(p))$ we have $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$.

OS-rule 23: For some X, p, we have $t = X$ and $X : p \in E$ and $\langle\, p, v \,\rangle\checkmark$. Note that $p = rhs(X)$. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$ and by definition of $F_E(X)$ we have $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$.

Theorem C.6 (Completeness: transitions)
Every transition of $\langle\, t \,|\, E \,\rangle$, is in $F_E(t)$. To prove:

$$\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle \Rightarrow \exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$$

Proof
Proof by induction on the length of the derivation of $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$. We make no distinction between base and step cases. We use case distinction on the OS-rule applied in the last step in the derivation. OS-rules 1, 4, 6, 8, 11, 13, 15, 22, 23 do not produce transitions so we only need to take the other rules into account. We use theorem 4.5 ("Completeness: termination").

OS-rule 2: $t = a$ for some action a and the definition of $F_E(a)$ combined with axiom 9-1 implies $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.

OS-rule 3: We have that t and t' are equal to a flow clause c. Thus $F_E(t) = F_E(c) = \{\langle\, [true] \gg c, c \,\rangle\}$. Since according to axiom 9-1 we have that $[true] \gg c \approx c$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.

OS-rule 5: We have for some d, p, p', v' that $t = d \gg p$ and $t' = p'$ and $(v, v') \models d$ and $\langle\, p, v' \,\rangle \rightarrow^l \langle\, p', v'' \,\rangle$. By induction we have that $\exists_{e,x,s'}(\langle\, e \gg x, t' \,\rangle \in F_E(p) \wedge \langle\, e \gg x, v' \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$. Therefore we have $\exists_{v'''}((v', v''') \models e)$ and thus $(v, v''') \models (d \sim e)$. Now by OS-rule 5 and the definition of $F_E(t)$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$

OS-rule 7: For some p, q we have that $t = p \oplus q$ and either $\langle\, p, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$ or $\langle\, q, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$. By induction we have $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(p) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$ or $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(q) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$. This implies according to the definition of $F_E(p \oplus q)$ that $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.

OS-rule 9: For some p, p', q we have $t = p \odot q$, $t' = p' \odot q$ and $\langle\, p, v \,\rangle \rightarrow^l \langle\, p', v'' \,\rangle$. By induction we have $\exists_{d,x,s'}(\langle\, d \gg x, p' \,\rangle \in F_E(p) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$. Therefore we have by definition of $F_E(p \odot q)$ that $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.

OS-rule 10: For some p, q we have $t = p \odot q$ and $\langle\, p, v \,\rangle\checkmark$ and $\langle\, q, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$. By theorem 4.5 we have $\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark)$. By OS-rule 4 we have $\exists_{v'}((v, v') \models d)$ and thus $(v, v) \models d^?$. By induction $\exists_{e,q'',s'}(\langle\, e \gg q'', t' \,\rangle \in F_E(q) \wedge \langle\, e \gg q'', v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.By OS-rule 5 $\exists_{v'''}((v, v'') \models e \wedge \langle\, q'', v''' \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle$.Thus by definition of $\sim$ we have $\langle\, d^? \sim e \gg q'', v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle$ and by definition of $F_E(p \odot q)$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.

OS-rule 12: For some p, p', q, $\langle\, p, v \,\rangle \rightarrow^l \langle\, p', v'' \,\rangle$, $t' = p' \blacktriangleright q$, and $t = p \blacktriangleright q$ or $t = p \rhd q$. By induction $\exists_{d,x,s'}(\langle\, d \gg x, p' \,\rangle \in F_E(p) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$. By definition of $F_E(p \blacktriangleright q)$ and $F_E(p \rhd q)$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.

OS-rule 14: For some p, q we have $t = p \blacktriangleright q$ and $\langle\, q, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$. By induction $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(q) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$ and thus by definition of $F_E(p \blacktriangleright q)$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$.

OS-rule 16: For some p, p', q, q', σ we have
$\langle\, p, v \,\rangle \rightsquigarrow^\sigma \langle\, p', v'' \,\rangle$ and $\langle\, q, v \,\rangle \rightsquigarrow^\sigma \langle\, q', v'' \,\rangle$ and $l = \sigma, t' = p' \parallel q'$ and either $t = p \parallel q$ or $t = p \,|\, q$. By induction $\exists_{d_1,c_1,s'}(\langle\, d_1 \gg c_1, p' \,\rangle \in F_E(p) \wedge \langle\, d_1 \gg c_1, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle)$ and $\exists_{d_2,c_2,u'}(\langle\, d_2 \gg c_2, t' \,\rangle \in F_E(t) \wedge \langle\, d_2 \gg c_2, v \,\rangle \rightsquigarrow^\sigma \langle\, u', v'' \,\rangle)$. By OS-rules 3 and 5 we have $v'' = \sigma(t)$ and $\exists_{v_1,v_2}((v, v_1) \models d_1, (v_1, \sigma) \models c_1, (v, v_2) \models d_2, (v_2, \sigma) \models c_2)$ and $dom(\sigma) = [0..t'']$ for some t. Now let $v' = \sigma(0)$. Since if $\exists_v((v, \sigma) \models c)$ then $(\sigma(0), \sigma) \models c$, we have that $(v', \sigma) \models c_1$ and $(v', \sigma) \models c_2$. Then by the definition of c_{jmp} we have that $(v_1, v') \models c_{1jmp}$ and $(v_2, v') \models c_{2jmp}$.

The definition of $d \sim d'$ gives us $(v, v') \models d_1 \sim c_{1jmp}$ and $(v, v') \models d_2 \sim c_{2jmp}$.
Combined, and using OS-rules 3 and 5, and the definition of $\wedge$ we get $\langle (d_1 \sim c_{1jmp}) \wedge (d_2 \sim c_{2jmp}) \gg c_1 \wedge c_2, v \rangle \rightsquigarrow^{\sigma} \langle c_1 \wedge c_2, v'' \rangle$. This is what $F_E(p \parallel q)$ and $F_E(p \mid q)$ return and therefore $\exists_{d,x,s'}(\langle d \gg x, t' \rangle \in F_E(t) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$.

OS-rule 17: For some p, p', q, σ we have $\langle p, v \rangle \rightsquigarrow^{\sigma} \langle t', v'' \rangle,\ \langle q, v \rangle\checkmark,\ l = \sigma,\ t' = p'$. Furthermore, $t = p \parallel q$ or $t = q \parallel p$ or $t = p \mid q$ or $t = q \mid p$.
By theorem 4.5 we have $\exists_e(\langle e \gg \epsilon, \checkmark \rangle \in F_E(q) \wedge \langle e \gg \epsilon, v \rangle\checkmark)$. By OS-rule 4 and the definition of ? we have $(v, v) \models e^{?}$. Induction: $\exists_{d,p'',s'}(\langle d \gg p'', p' \rangle \in F_E(p) \wedge \langle d \gg p'', v \rangle \rightsquigarrow^{\sigma} \langle s', v'' \rangle)$. OS-rule 5 gives $\exists_{v'}((v, v') \models d \wedge \langle p'', v' \rangle \rightsquigarrow^{\sigma} \langle s', v'' \rangle$. By definition of $F_E(p \parallel q)$ and $F_E(p \mid q)$ and OS-rule 5 we get $\exists_{d,x,s'}(\langle d \gg x, t' \rangle \in F_E(t) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$.

OS-rule 18: For some p, p', q, v' we have $l = \{a, v'\}$ and $\langle p, v \rangle \rightarrow^{a,v'} \langle p', v \rangle$. Furthermore, we have either ($t = p \parallel q$ and $t' = p' \parallel q$) or ($t = q \parallel p$ and $t' = q \parallel p'$) or ($t = p \lfloor\!\lfloor q$ and $t' = p' \parallel q$).
By induction $\exists_{d,x,s'}(\langle d \gg x, p' \rangle \in F_E(p) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$. Since $l = \{a, v'\}$, we have by OS-rules 1,2,3,4,5 that $x = a$. Therefore, by the definition of $F_E(p \parallel q)$ and $F_E(p \lfloor\!\lfloor q)$ we have $\exists_{d,x,s'}(\langle d \gg x, t' \rangle \in F_E(t) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$.

OS-rule 19: For some $p, p', q, q', a_1, a_2, a, v'$ we have $l = \{a, v'\}$ and $\langle p, v \rangle \rightarrow^{a_1,v'} \langle p', v \rangle$ and $\langle q, v \rangle \rightarrow^{a_2,v'} \langle q', v \rangle$ and $\gamma(a_1, a_2) = a$. Furthermore, $t = p \parallel q$ or $t = p \mid q$ and $t' = p' \parallel q'$.
By induction and OS-rule 1,2,3,4,5 we have $\exists_{d_1,s'}(\langle d_1 \gg a_1, p' \rangle \in F_E(p) \wedge \langle d_1 \gg a_1, v \rangle \rightarrow^{a_1,v} \langle s', v'' \rangle)$ and $\exists_{d_2,s'}(\langle d_2 \gg a_2, q' \rangle \in F_E(q) \wedge \langle d_2 \gg a_2, v \rangle \rightarrow^{a_2,v'} \langle s', v'' \rangle)$ By OS-rule 5 and the definition of $\wedge$ we get that $\langle d_1 \wedge d_2 \gg a, v \rangle \rightarrow^{a,v'} \langle \epsilon, v'' \rangle$. Therefore, by definition of $F_E(p \parallel q)$ and $F_E(p \mid q)$ we have $\exists_{d,x,s'}(\langle d \gg x, t' \rangle \in F_E(t) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$.

OS-rule 20: For some p, p', a, v' we have $l = \{a, v'\}$ and $\langle p, v \rangle \rightarrow^{a,v'} \langle p', v \rangle$ and $t = \partial_H(p)$ and $t' = \partial_H(p')$ and $a \notin H$. By induction and OS-rules 1,2,3,4,5 $\exists_{d,s'}(\langle d \gg a, p' \rangle \in F_E(p) \wedge \langle d \gg a, v \rangle \rightarrow^{a,v} \langle s', v'' \rangle)$. Thus, by the definition of $F_E(\partial_H(p))$ we have $\exists_{d,x,s'}(\langle d \gg x, t' \rangle \in F_E(t) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$.

OS-rule 21: For some p, p', σ we have $\langle p, v \rangle \rightsquigarrow^{\sigma} \langle p', v'' \rangle$ and $t = \partial_H(p), t' = \partial_H(p'), l = \sigma$. By induction and OS-rules 1,2,3,4,5 we have $\exists_{d,c,s'}(\langle d \gg c, p' \rangle \in F_E(p) \wedge \langle d \gg c, v \rangle \rightsquigarrow^{\sigma} \langle s', v'' \rangle)$. Therefore, by definition of $F_E(\partial_H(p))$ we have $\exists_{d,x,s'}(\langle d \gg x, t' \rangle \in F_E(t) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$.

OS-rule 24: For some X, p we have $t = X$ and $X : p \in E$ and $\langle p, v \rangle \rightarrow^{l} \langle t', v'' \rangle$. Note that $p = rhs(X)$. By induction $\exists_{e,x,s'}(\langle e \gg x, t' \rangle \in F_E(q) \wedge \langle e \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$. Thus by definition of $F_E(X)$ we have $\exists_{d,x,s'}(\langle d \gg x, t' \rangle \in F_E(t) \wedge \langle d \gg x, v \rangle \rightarrow^{l} \langle s', v'' \rangle)$.

C.4 Renaming operators

All proofs related to the extension of $F_E(t)$ with renaming operators are here. It turns out that we need to assume that the solutions of a clause are not dependent on the name of the model variables, i.e. $(v, v') \models d$ iff $(V(v), V(v')) \models V(d)$, and $(v, \sigma) \models c$ iff $(V(v), V(\sigma)) \models V(c)$. This is a (reasonable) restriction on the clause formalism.

Theorem C.7 (Well-definedness)
The extended version of $F_E(t)$ is well-defined for substitutably guarded specifications.

Proof
Take the weight function $w(t)$ from theorem 4.2. In the new $F_E(t)$-rules, only $F_E(p)$ is evaluated. This decreases the number of operators as well as the number of (possibly unguarded) recursion

variables. Therefore, $w(\langle p|E\rangle) < w(t|E\rangle)$ in each rule.

Theorem C.8 (Form of $F_E(t)$)
The form of $F_E(t)$ as shown in theorem 4.1 is left intact by the new rules.

Proof
By construction.

Theorem C.9 (Soundness: termination)
Termination in the extended version of $F_E(t)$ implies that $\langle\, t \,|\, E \,\rangle$ can terminate. To prove:

$$\exists_d(\langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v \,\rangle\checkmark) \Rightarrow \langle\, t, v \,\rangle\checkmark$$

Proof
Extend the proof of theorem 4.3 with the following:

Rule 15: $t = \rho_A(p),\ \langle\, d \gg \epsilon, \checkmark \,\rangle \in F_E(p)$ for some p. By induction and OS-rule 25.

Rule 16: $t = \varrho_V(p),\ d = V(d'),\ \langle\, d' \gg \epsilon, \checkmark \,\rangle \in F_E(p)$ for some p, d'. From $\langle\, d \gg \epsilon, v \,\rangle\checkmark$ we conclude $\exists_{v'}((v, v') \models_{\mathrm{r}} d)$. By the extra assumption, $(v, v') \models_{\mathrm{r}} d'$. By induction, $\langle\, p, v \,\rangle\checkmark$. By OS-rule 28, $\langle\, t, v \,\rangle\checkmark$.

Theorem C.10 (Soundness: transitions)
All transitions in the extended version of $F_E(t)$ can be performed by $\langle\, t \,|\, E \,\rangle$. To prove:

$$\exists_{d,x,s'}(\langle\, d \gg x, t' \,\rangle \in F_E(t) \wedge \langle\, d \gg x, v \,\rangle \rightarrow^l \langle\, s', v'' \,\rangle) \Rightarrow \langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$$

Proof
Extend the proof of theorem 4.4 with the following:

Rule 15: $t = \rho_A(p),\ t' = \rho_A(p'')$ for some p, p', A. We have two cases.

- Case 1: $x = A(a),\ \langle\, d \gg a, p'' \,\rangle \in F_E(p)$ for some a. By OS-rules 2 and 5, we have that $l = (A(a), v'')$ and $(v, v'') \models d$. Again by OS-rules 2 and 5, $\langle\, d \gg a, v \,\rangle \rightarrow^{(a,v'')} \langle\, \epsilon, v'' \,\rangle$. By induction we have that $\langle\, p, v \,\rangle \rightarrow^{(a,v'')} \langle\, p'', v'' \,\rangle$. By OS-rule 26, we get $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.
- Case 2: $x = c,\ \langle\, d \gg x, p'' \,\rangle \in F_E(p)$ for some c. By OS-rules 5 and 3, we have that $l = \sigma$ for some σ. By induction and OS-rule 27 we get $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.

Rule 16: $t = \varrho_V(p),\ d = V(d'),\ t' = \varrho_V(p'')$ for some p, p'', d'. We have two cases.

- Case 1: $x = a,\ \langle\, d' \gg a, p'' \,\rangle \in F_E(p)$ for some a. By OS-rules 2 and 5, we have that $l = (a, v'')$ and $(v, v'') \models d$. By the extra assumption we have that $(V^{-1}(v), V^{-1}(v'')) \models d'$ and thus $\langle\, d' \gg a, V^{-1}(v) \,\rangle \rightarrow^{(a,V^{-1}(v))} \langle\, \epsilon, V^{-1}(v'') \,\rangle$. By induction we get $\langle\, p, V^{-1}(v) \,\rangle \rightarrow^{(a,V^{-1}(v))} \langle\, p'', V^{-1}(v'') \,\rangle$. By OS-rule 29 we get $\langle\, t, v \,\rangle \rightarrow^l \langle\, t', v'' \,\rangle$.
- Case 2: $x = V(c),\ \langle\, d' \gg c, p'' \,\rangle \in F_E(p)$ for some a. By OS-rules 3 and 5, we have that $l = \sigma$ and $(v, v') \models d,\ (v', \sigma) \models V(c),\ dom(\sigma) = [0..t''],\ v'' = \sigma(t'')$ for some v', σ, t''. By the extra assumption we have that $(V^{-1}(v), V^{-1}(v')) \models d',\ (V^{-1}(v'), V^{-1}(\sigma)) \models c$ and

thus $\langle\, d' \gg c, V^{-1}(v)\,\rangle \rightsquigarrow^{V^{-1}(\sigma)} \langle\, c, V^{-1}(\sigma(t''))\,\rangle$. By induction we get $\langle\, p, V^{-1}(v)\,\rangle \rightsquigarrow^{V^{-1}(\sigma)} \langle\, p'', V^{-1}(v'')\,\rangle$. By OS-rule 30 we get $\langle\, t, v\,\rangle \rightarrow^{l} \langle\, t', v''\,\rangle$.

Theorem C.11 (Completeness: termination)
Termination of $\langle\, t \mid E\,\rangle$ is in the extended version of $F_E(t)$. To prove:

$$\langle\, t, v\,\rangle\checkmark \Rightarrow \exists_d(\langle\, d \gg \epsilon, \checkmark\,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v\,\rangle\checkmark)$$

Proof
Extend the proof of theorem 4.5 with the following:

OS-rule 25: $t = \rho_A(p)$ for some A, p and $\langle\, p, v\,\rangle\checkmark$. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark\,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v\,\rangle\checkmark)$. Therefore by definition of $F_E(\rho_A(p))$ we have $\exists_d(\langle\, d \gg \epsilon, \checkmark\,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v\,\rangle\checkmark)$.

OS-rule 28: $t = \varrho_V(p)$ for some V, p and $\langle\, p, v\,\rangle\checkmark$. By induction $\exists_d(\langle\, d \gg \epsilon, \checkmark\,\rangle \in F_E(p) \wedge \langle\, d \gg \epsilon, v\,\rangle\checkmark)$. Therefore by definition of $F_E(\varrho_V(p))$ we have $\exists_d(\langle\, d \gg \epsilon, \checkmark\,\rangle \in F_E(t) \wedge \langle\, d \gg \epsilon, v\,\rangle\checkmark)$.

Theorem C.12 (Completeness: transitions)
Every transition of $\langle\, t \mid E\,\rangle$, is in the extended version of $F_E(t)$. To prove:

$$\langle\, t, v\,\rangle \rightarrow^{l} \langle\, t', v''\,\rangle \Rightarrow \exists_{d,x,s'}(\langle\, d \gg x, t'\,\rangle \in F_E(t) \wedge \langle\, d \gg x, v\,\rangle \rightarrow^{l} \langle\, s', v''\,\rangle)$$

Proof
Extend the proof of theorem 4.6 with the following:

OS-rule 26: $t = \rho_A(p)$, $t' = \rho_A(p')$, $l = (A(a), v')$ for some a, v', A, p, p' and $\langle\, p, v\,\rangle \rightarrow^{(a,v')} \langle\, p', v''\,\rangle$. By induction $\exists_{d,x,s'}(\langle\, d \gg x, p'\,\rangle \in F_E(p) \wedge \langle\, d \gg x, v\,\rangle \rightarrow^{(a,v')} \langle\, s', v''\,\rangle)$. By the form of $F_E(t)$ and OS-rules 1 through 5, we have that $x = a$. By OS-rules 2 and 5 we have that $\langle\, d \gg A(a), v\,\rangle \rightarrow^{(A(a),v')} \langle\, s', v''\,\rangle)$. Therefore, by definition of $F_E(\rho_A(p))$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t'\,\rangle \in F_E(t) \wedge \langle\, d \gg x, v\,\rangle \rightarrow^{l} \langle\, s', v''\,\rangle)$.

OS-rule 27: $t = \rho_A(p)$, $t' = \rho_A(p')$, $l = \sigma$ for some a, A, p, p', σ and $\langle\, p, v\,\rangle \rightsquigarrow^{\sigma} \langle\, p', v''\,\rangle$. By induction $\exists_{d,x,s'}(\langle\, d \gg x, p'\,\rangle \in F_E(p) \wedge \langle\, d \gg x, v\,\rangle \rightsquigarrow^{\sigma} \langle\, s', v''\,\rangle)$. Therefore by definition of $F_E(\rho_A(p))$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t'\,\rangle \in F_E(t) \wedge \langle\, d \gg x, v\,\rangle \rightarrow^{l} \langle\, s', v''\,\rangle)$.

OS-rule 29: $t = \varrho_V(p)$, $t' = \varrho_V(p')$, $l = (a, V(v'))$ for some a, v', V, p, p' and $\langle\, p, v\,\rangle \rightarrow^{(a,v')} \langle\, p', v''\,\rangle$. By induction $\exists_{d,x,s'}(\langle\, d \gg x, p'\,\rangle \in F_E(p) \wedge \langle\, d \gg x, v\,\rangle \rightarrow^{(a,v')} \langle\, s', v''\,\rangle)$. By the form of $F_E(t)$ and OS-rules 1 through 5, we have that $x = a$. By OS-rules 2, 5 and the extra assumption, we have that $\langle\, V(d) \gg a, v\,\rangle \rightarrow^{(a,V(v))} \langle\, s', V(v'')\,\rangle)$. Therefore, by definition of $F_E(\varrho_V(p))$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t'\,\rangle \in F_E(t) \wedge \langle\, d \gg x, v\,\rangle \rightarrow^{l} \langle\, s', v''\,\rangle)$.

OS-rule 30: $t = \varrho_V(p)$, $t' = \varrho_V(p')$, $l = V(\sigma)$ for some σ, V, p, p' and $\langle\, p, v\,\rangle \rightsquigarrow^{\sigma} \langle\, p', v''\,\rangle$. By induction $\exists_{d,x,s'}(\langle\, d \gg x, p'\,\rangle \in F_E(p) \wedge \langle\, d \gg x, v\,\rangle \rightsquigarrow^{\sigma} \langle\, s', v''\,\rangle)$. By the form of $F_E(t)$ and OS-rules 1 through 5, we have that $x = c$ for some flow clause c. By OS-rules 3, 5 and the extra assumption, we have that $\langle\, V(d) \gg c, V(v)\,\rangle \rightsquigarrow^{V(\sigma)} \langle\, s', V(v'')\,\rangle)$. Therefore, by definition of $F_E(\varrho_V(p))$ we have $\exists_{d,x,s'}(\langle\, d \gg x, t'\,\rangle \in F_E(t) \wedge \langle\, d \gg x, v\,\rangle \rightarrow^{l} \langle\, s', v''\,\rangle)$.

Appendix D

Rewriting flows

In section 4.1.3.1, c_{jmp} constructs are eliminated by rewriting flows. This appendix gives the proof of

$$\left(\; V \;\middle|\; pred \; \right) \leftrightarrows_r X$$

where

$$X : \left[\; \mathcal{V}_{\mathrm{m}} \backslash V \;\middle|\; true \; \right] \gg \left(\; \mathcal{V}_{\mathrm{m}} \;\middle|\; pred \; \right) \rhd X$$

Abbreviations

We introduce the following abbreviations:

$$\begin{array}{lll} c_o & \text{for} & \left(\; V \;\middle|\; pred \; \right) \\ c_v & \text{for} & \left(\; \mathcal{V}_{\mathrm{m}} \;\middle|\; pred \; \right) \\ d_v & \text{for} & \left[\; \mathcal{V}_{\mathrm{m}} \backslash V \;\middle|\; true \; \right] \end{array}$$

Proof outline

We prove that $X \to c_o$ is a solution for $X : d_v \gg c_v \rhd X$. Trivially, $X \to X$ is a solution as well. Because $d_v \gg c_v \rhd X$ is guarded, we may apply the Recursive Specification Principle and we get $c_o \approx_r X$. By the soundness of $\approx_r$ we get that $c_o \leftrightarrows_r X$.

Proof: $X \to c_o$ is a solution

For this we must prove that $c_o \leftrightarrows_r d_v \gg c_v \rhd c_o$. Interestingly, this cannot be proven using the HyPA derivation rules. Therefore, we prove this by giving a witnessing robust bisimilation relation R:

$$\forall_{v \in Val, V \subseteq \mathcal{V}_{\mathrm{m}}, pred \in P_{\mathrm{f}}} \left(\; \langle\, c_o, v \,\rangle R \langle\, c_o, v \,\rangle \wedge \langle\, c_o, v \,\rangle R \langle\, c_v \blacktriangleright c_o, v \,\rangle \wedge \langle\, c_o, v \,\rangle R \langle\, d_v \gg c_v \rhd c_o, v \,\rangle \; \right)$$

Proof: R is a robust bisimilation relation

Now we prove that R is indeed a robust bisimilation relation. R is robust by construction. As for bisimilarity, we can skip the termination cases since none of the given processes can terminate. For the other cases, any transition is a flow transition. Lemmas D.1 and D.2 prove these cases.

Lemma D.1 (pRq and $\langle\, p, v \,\rangle \rightsquigarrow^{\sigma} \langle\, p', v' \,\rangle$ implies $\exists_{q'}(\langle\, q, v \,\rangle \rightsquigarrow^{\sigma} \langle\, q', v' \,\rangle$ and $p' R q')$)
We have three cases. In all cases, $p \equiv c_o$ by definition of R, and $p' \equiv c_o$ by the operational semantics.
Case 1: $q \equiv c_o$. Trivial.

Case 2: $q \equiv c_v \blacktriangleright c_o$.
By OS-rule 14 we get $\langle q, v \rangle \rightsquigarrow^{\sigma} \langle c_o, v' \rangle$. By definition of R we also have $p' R q'$.
Case 3: $q \equiv d_v \gg c_v \rhd c_o$.
By OS-rule 5 and the definition of the solution of a flow clause we have $(v, \sigma) \models_f c_o$ and $\forall_{x \in V}(v(x) = \sigma(0)(x))$. This gives us $(v, \sigma(0)) \models_r d_v$ and $(\sigma(0), \sigma) \models_f c_v$. By OS-rules 3, 5, and 12 we get $\langle q, v \rangle \rightsquigarrow^{\sigma} \langle c_v \blacktriangleright c_o, v' \rangle$. By definition of R, $p' R q'$.

Lemma D.2 (*pRq* and $\langle q, v \rangle \rightsquigarrow^{\sigma} \langle q', v' \rangle$ implies $\exists_{p'}(\langle p, v \rangle \rightsquigarrow^{\sigma} \langle p', v' \rangle$ and $p' R q')$)
We have three cases. In all cases, $p \equiv c_o$ by definition of R, and $p' \equiv c_o$ by the operational semantics.
Case 1: $q \equiv c_o$ Trivial.
Case 2: $q \equiv c_v \blacktriangleright c_o$
We have two cases: the transition from $\langle q, v \rangle$ can be according to OS-rule 12 or to OS-rule 14.
Case 2.1: OS-rule 12: $\langle c_v, v \rangle \rightsquigarrow^{\sigma} \langle q'', v' \rangle$
By OS-rule 5 we have that $q'' \equiv c_v$ (and thus $q' \equiv c_v \blacktriangleright c_o$) and $(v, \sigma) \models c_v$. Therefore $\sigma \models_f \mathit{pred}$ and $\forall_{x \in \mathcal{V}_\mathrm{m}}(v(x) = \sigma(0)(x))$. Since $V \subseteq \mathcal{V}_\mathrm{m}$ we also have $\forall_{x \in V}(v(x) = \sigma(0)(x))$ and thus $(v, \sigma) \models c_o$. Thus, $\langle p, v \rangle \rightsquigarrow^{\sigma} \langle p', v' \rangle$. By definition of R, $p' R q'$.
Case 2.2: OS-rule 14: $\langle c_o, v \rangle \rightsquigarrow^{\sigma} \langle q', v' \rangle$
By OS-rule 5 we have that $q' \equiv c_o$. Trivially, $\langle p, v \rangle \rightsquigarrow^{\sigma} \langle p', v' \rangle$ and $p' R q'$.
Case 3: $q \equiv d_v \gg c_v \rhd c_o$.
By OS-rules 12, 5, 3 and the definitions of clause solutions we have $(v, \sigma(0)) \models_r \mathit{true}$ and $\forall_{x \in V}(v(x) = \sigma(0)(x))$ and $(\sigma(0), \sigma) \models_f \mathit{pred}$ and $q' \equiv c_v \blacktriangleright c_o$. This implies that $(v, \sigma) \models_f c_o$. By OS-rule 5 we get $\langle p, v \rangle \rightsquigarrow^{\sigma} \langle p', v' \rangle$ and by definition of R, $p' R q'$.

Appendix E

Screenshots

This appendix contains screenshots of the various prototypes and simulator versions.

Figure E.1 shows the first prototype: a dummy user interface with every visualization element in its own window. There is a window containing the HyPA model to be simulated, a list of all transitions that have been performed, two X/T plots showing the value of variables X and Y over time. On the upper-right there is the main window with all controls needed to run the simulation, open files etcetera. The list on the left of the main window is the list of current choices. This window also contains a plot with all variables. In the real version, this plot could be used to select the time length of a flow clause to perform.

The second prototype is in figure E.2. The entire user interface is contained within one window. X/T plots are shown below each other, and each X/T plot displays the same time interval. This makes the relations between the variables more clear.

Figure E.3 shows a working version of the HyPA simulator. Simulation controls are in a toolbar at the top. On the upper left we see the currently possible transitions. The upper right section shows actions that have occurred, and the plots show the continuous behavior. On the lower left is a legend showing the color used for each variable, and the current value of the variables.

Figure E.1: Prototype 1 user interface

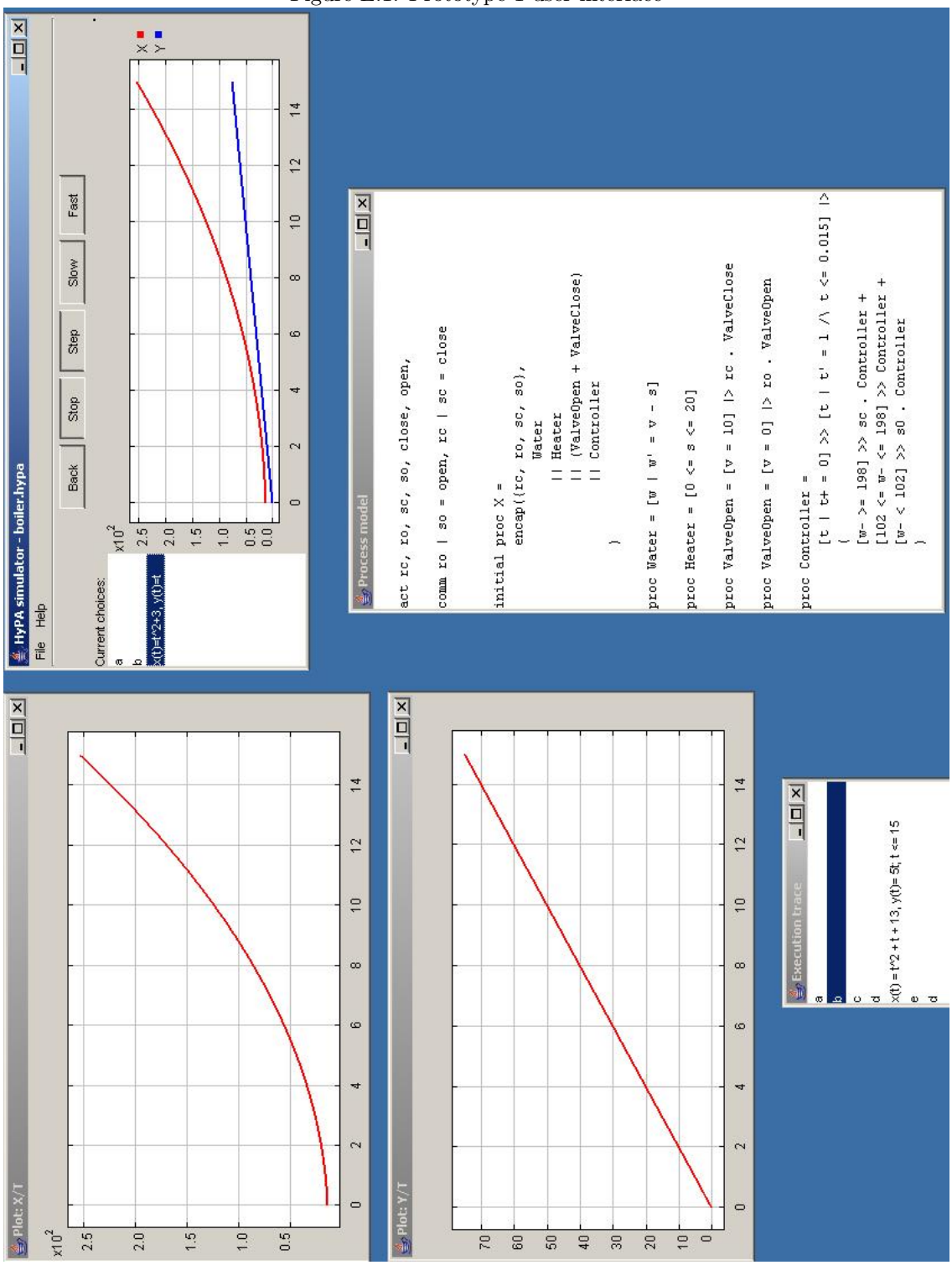


Figure E.2: Prototype 2 user interface

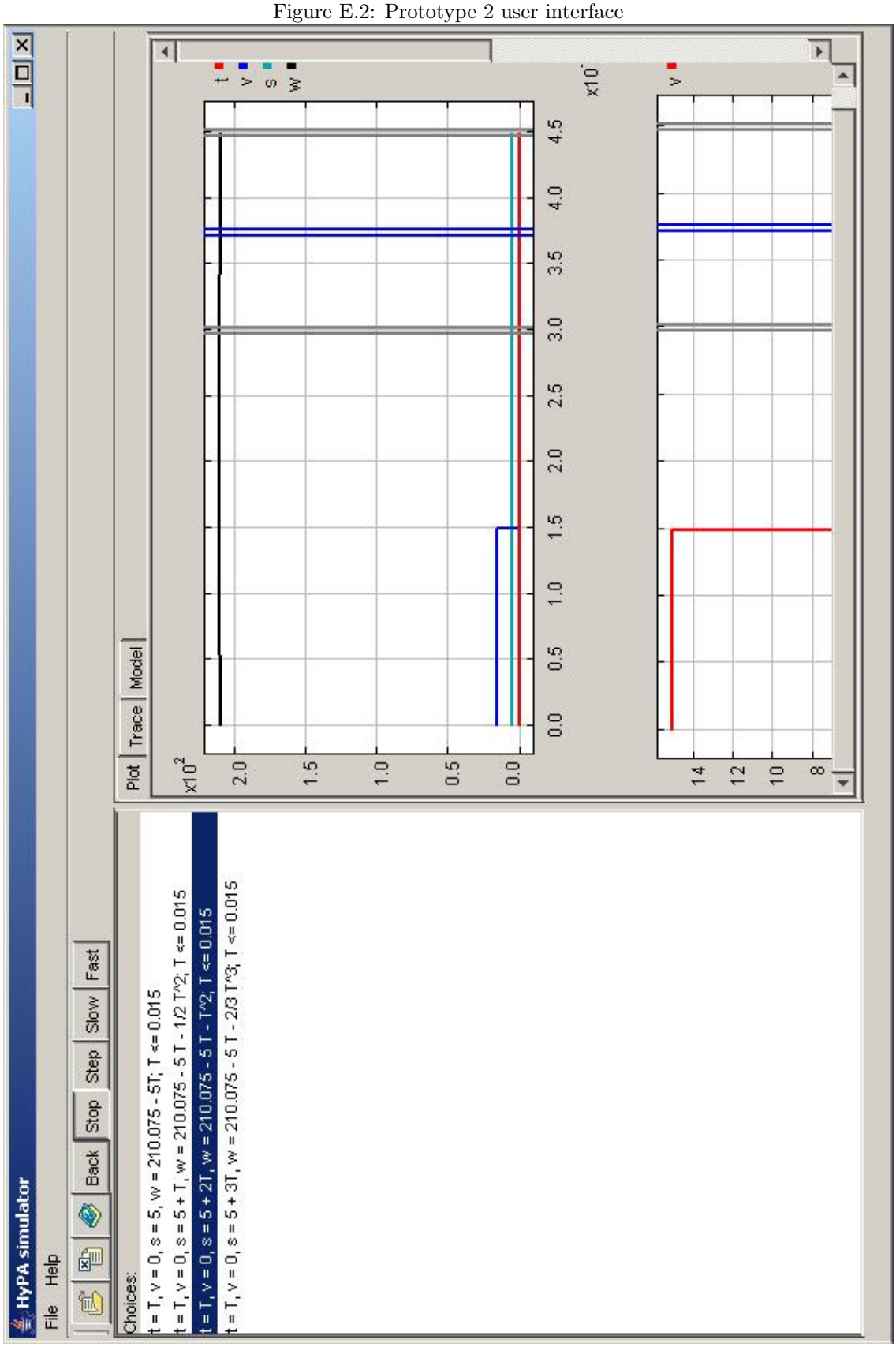
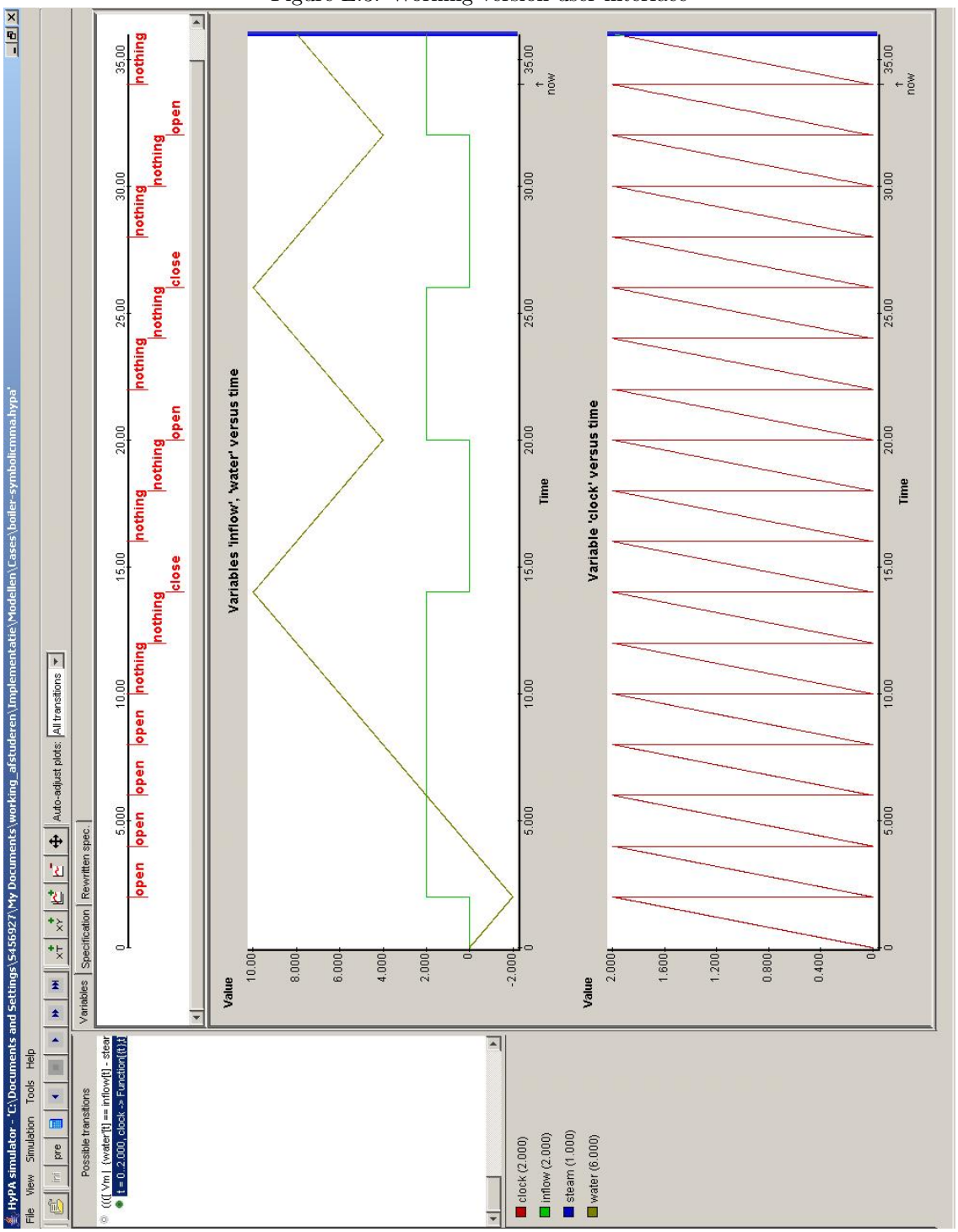


Figure E.3: Working version user interface



Appendix F

User manual

This chapter describes the installation and use of the HyPA simulator.

F.1 Installation

The following is needed in order to run the HyPA simulator:

- A computer running Windows XP;
- Mathematica version 5.1;
- The Java Runtime Environment (JRE) version 1.4.2 or later;
- File "hypasim.jar";
- File "hypasimW.exe";
- File "hypasim.exe" (optional);
- File "JLinkNativeLibrary.dll".

The JRE can be downloaded free at http://java.com. The file "JLinkNativeLibrary.dll" must be placed in the jre\bin directory. The other two files must be in a directory together.

F.2 Running

Running the simulator simply amounts to running "hypasimW.exe". If the simulator does not seem to be working, you might want to run "hypasim.exe" (without a 'W') from a console (DOS box). That way, you can see any Java-specific errors that might occur.

There is a setting that needs to be adjusted before any models can be simulated. This setting can be found in the **Tools** menu, under **Options**. You are presented with a dialog with three tabs. Choose the **Mathematica** tab. In the field on that tab, enter the path to the mathematica kernel. Usually, you can leave the default value of
`C:\Program Files\Wolfram Research\Mathematica\5.1\MathKernel.exe`.

F.3 User interface

Figure F.1 shows the HyPAsim user interface. Five main components are indicated with red arrows:

1. The *toolbar*, containing buttons for controlling the simulation and visualizations;

Figure F.1: HyPAsim user interface

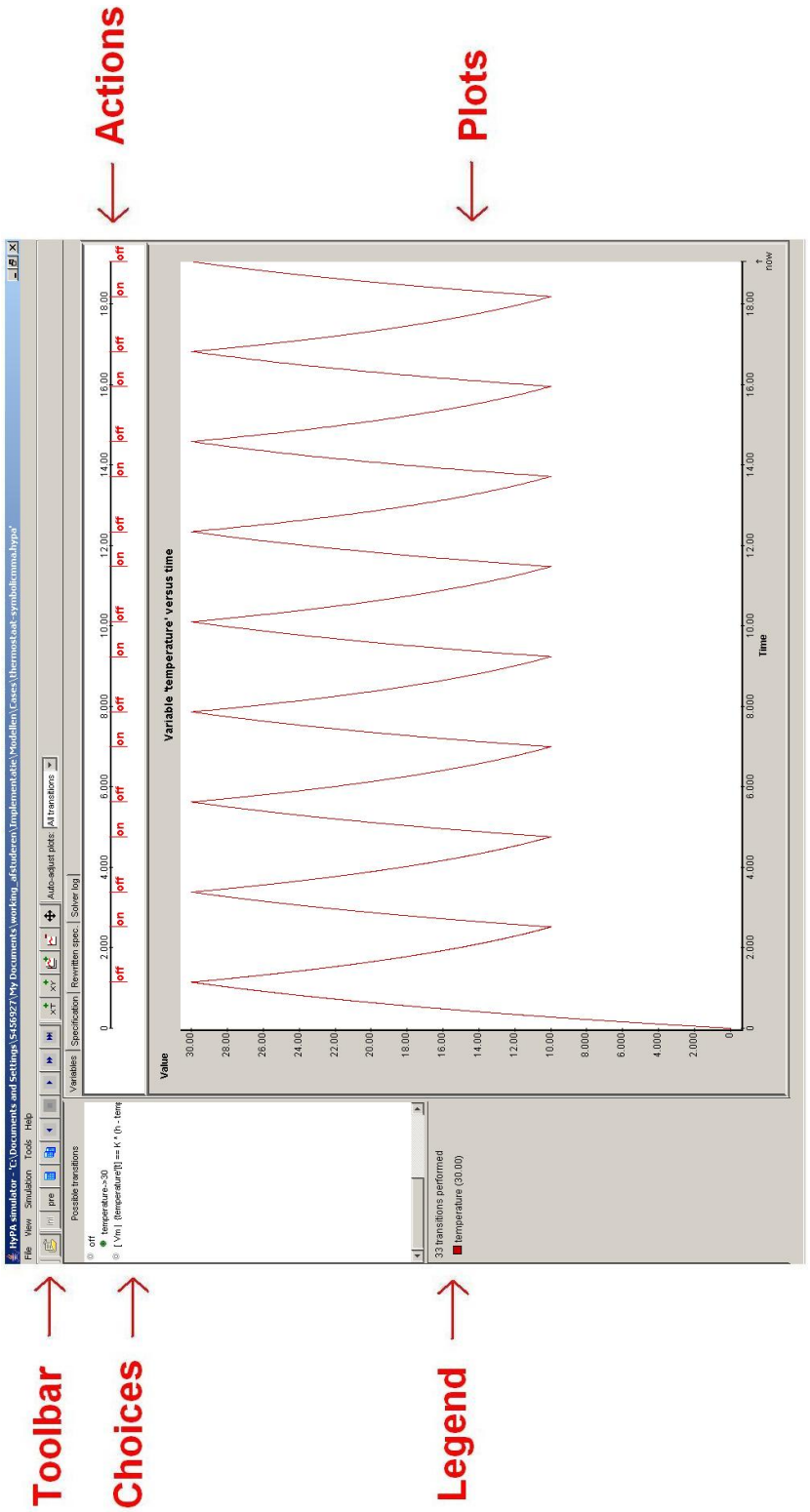


Figure F.2: HyPAsim toolbar

2. The *choices* box, where all currently possible transitions and their solutions are shown;
3. The *legend*, showing the color that each model variable has in the plots, and the current value of each model variable;
4. The *actions* viewer, showing the performed actions against a time-line;
5. The *plots* pane, where the model variable behavior is shown in X/T or X/Y plots.

Figure F.2 gives a closer view of the toolbar. Note that during actual simulation, some buttons may be grayed out to indicate that they cannot be pressed at this time. Here is a short description per button; a longer explanation is in the next sections.

1. Load a HyPA model from a file;
2. Enter initial values for the model variables;
3. Enter additional predicates for the selected transition;
4. Solve the selected transition;
5. Solve all transitions;
6. Undo the last performed transition;
7. Stop random simulation;
8. Perform the selected transition;
9. Start random simulation;
10. Start random simulation until a stop criterium is reached;
11. Add an X/T plot;
12. Add an X/Y plot;
13. Add an X/T plot for each model variable;
14. Remove one or more plots;
15. Fit all plot axis ranges so that everything is visible;
16. Settings for auto-adjustment of plots during simulation.

F.4 Loading a HyPA model

To load a HyPA model, press button **1**. You are presented with a file chooser dialog. The simulator assumes that HyPA model files have extension `.hypa`. Choose a file and click **OK**. Now several things happen:

- Dependent on the clause formalism used in the model, Mathematica is started (visible in the Windows task bar);
- One X/T plot is shown that has all the model variables in it;
- The legend shows all model variables in the model;
- The simulator starts solving the transitions that the model can do. It will do this until all first transitions are solved, or until a timeout expires (see below, F.8).

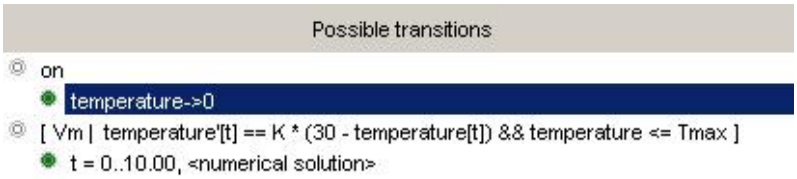
You can view the model file contents on the **Specification** tab above the action viewer. Internally, the simulator uses a rewritten version of the model (see section 4.1.3.1). This rewritten version is under the **Rewritten spec.** tab.

Because no transitions have been performed yet, you have the option of setting initial values for the model variables. You can do so by pressing button **2**. How the initial values should be entered, is clause formalism-specific.

F.5 Step-wise simulation

In step-wise simulation, you are presented with a choice after each transition. The simulator finds out which transitions the model can do, and displays them in the choice box. Also, it tries to solve the clauses in these transitions for a while. The time that the simulator spends on solving is a setting (section F.8). The solutions are also shown in the choice box. Figure F.3 shows two transitions that each have one solution. The transitions are preceded by a grey/white dot, and the solutions by a green one. The first transition in the figure is an action transition, the second one a flow transition.

Figure F.3: Choice box



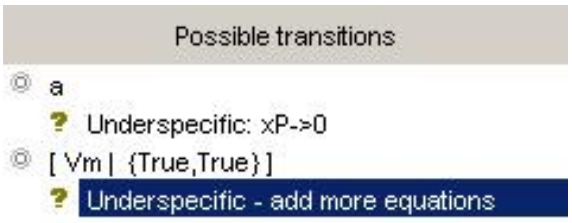
In section 4.1.1, it is explained that first transitions are of the form $d \gg a$, $d \gg c$ or $d \gg \epsilon$. The simulator initially only shows the a, c, or ϵ. Also, there is a set of next processes per transition. For each next process, a separate transition is shown in the choice box. You can right-click on the choice box to show the reinitialization and the next processes.

F.5.1 Solving transitions

If a transition is not solved (due to the time limit), it is shown without solutions. You can select such a transition and press button **4** to solve it manually. Alternatively, you can press button **5**

to solve all transitions.

Figure F.4: Underspecific clauses



Sometimes, the clauses in a model are not specific enough to be solved. For instance, the flow clause $\left(\; s \;\middle|\; s_{min} \leq s \leq s_{max} \; \right)$ does not specify actual solutions for s. In this case, solutions are shown in the choice box with a preceding question mark, like in figure F.4. You can double-click the solutions, or press button **3**, to add predicates that make the clauses more precise.

F.5.2 Choosing a solution

From the solutions in the choice box, you have to choose one to perform. You do this by clicking it. Note that all visualizations show the past behavior, *and* the selected solution. In case of a flow, you also have to choose a time length for the transition. The X/T plots will show a blue vertical bar. Using the mouse, you can drag this bar left and right to select the length of the flow.

Once you have chosen a solution (and possibly a time length), you can click button **8** to perform it. Double-clicking the solution also works.

In case of deadlock, the simulator presents an empty choice box.

F.6 Random simulation

The simulator can make all choices on its own. You can start random simulation by clicking button **9**. Most buttons except for the stop-button (number **7**) are grayed out. The visualizations are updated periodically. The frequency of these updates is once a second, but can be delayed somewhat by lengthy calculations in Mathematica. The random simulation stops in the following cases:

- Deadlock or termination;
- No solutions, or only under-specific solutions found;
- User abort using button **7**.

Alternatively, you can start random simulation with button **10**. You are first presented with a dialog that allows you to specify a stop criterium. In addition to the cases above, random simulation will also stop on reaching the stop criterium.

During random simulation, the user interface becomes a bit sluggish. However, all non-grayed elements can be manipulated, with some patience.

F.7 Visualizations

Model behavior is visualized with the action viewer, X/T plots, and X/Y plots.

Figure F.5: Action viewer

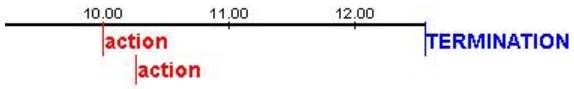
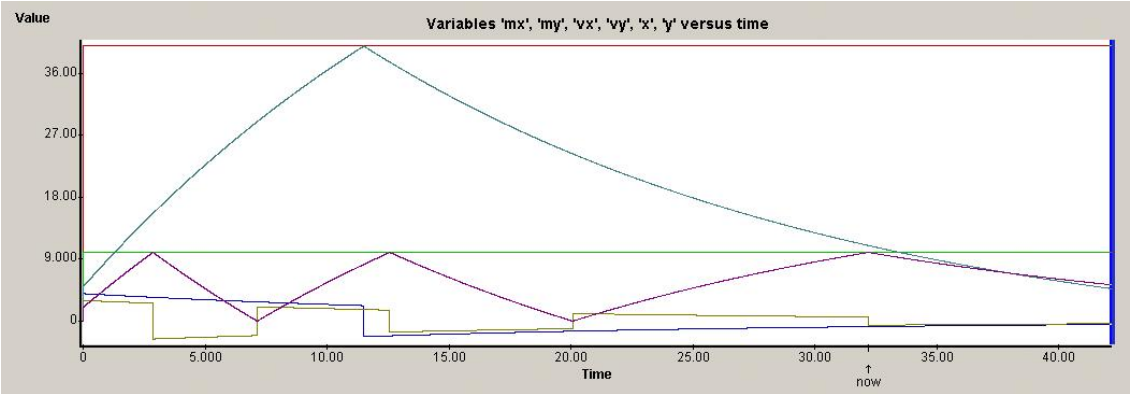


Figure F.6: X/T plot



F.7.1 Action viewer

The action viewer (figure F.5) is a fixed element: you cannot remove or add action viewers. It remains at the top of the screen. It shows actions against a time line. Actions are in red. The range of the time line is kept in sync with the time-axis of any X/T plots. That means that zooming/scrolling in the X/T plots (described below) also affects the action viewer.

When there are too many actions to fit next to each other, actions are shown below each other. In case of successful termination, the blue word "TERMINATION" is shown in the action viewer.

F.7.2 Plots

There are currently two types of plots: variables versus time ("X/T"), and two variables versus one another ("X/Y"). You can add and remove plots at any time during the simulation. One can add plots by clicking button **11** (add X/T plot), button **12** (add X/Y plot) or button **13** (add X/T plot for each variable). In the first two cases, you are presented with a dialog to choose variables. Removing plots is done using button **14**. You can select one or more plots to remove in the dialog that opens.

F.7.2.1 X/T plots

An X/T plot (figure F.6) shows the value of one or more variables against time. The variables are mentioned in the plot title at the top. The blue bar at the right is the time selection bar that is mentioned in section F.5.2. You can drag it left and right to select the length of a flow transition. A label "now" (detail view in figure F.7) shows the current time. Anything to the left of "now" is from performed transitions, and anything to the right is a transition that is currently selected in the choice box.

Figure F.7: X/T plot: current time

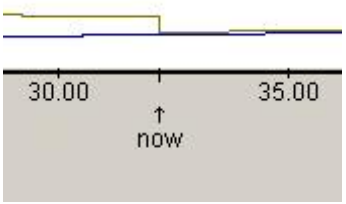
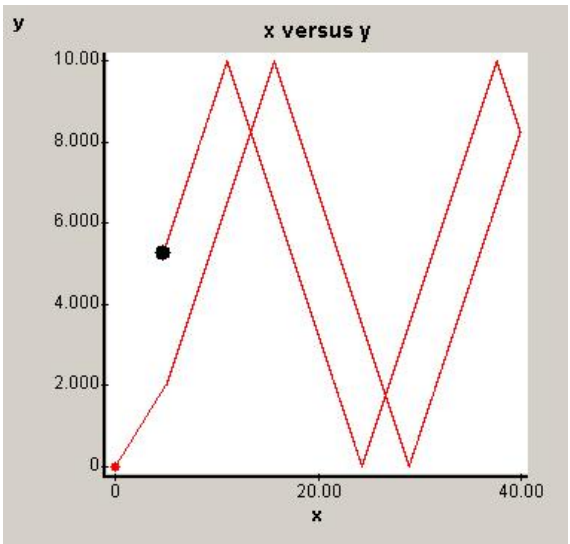


Figure F.8: X/Y plot



F.7.2.2 X/Y plots

An X/Y plot (figure F.8) shows the value of two variables versus one another. The initial values are represented by a red dot, and the last values are represented by a black dot.

F.7.2.3 Zooming and scrolling

You can *zoom in* on a portion of a plot by dragging your mouse across the area, in a **down-right** direction. There may be a short delay before the zoomed area is visible, because the simulator might have to ask Mathematica for more plot points.

Zooming out is done by dragging the mouse in the opposite, **up-left** direction. During the drag, you see two overlapping blue boxes. The inner box represents the current view, and the outer box represents the new view. The difference between the old and new view is proportional to the difference between the boxes. You can only zoom out until all the data in the plot is shown. If you want to zoom out entirely, you can also use button **15**.

You can *scroll* through a plot by dragging your mouse using the **right mouse button**. You can drag in any direction, and the plot will stick to your mouse pointer.

Note that the time-axis ranges on all plots and on the action viewers are synchronized. When you scroll through one plot, all other plots will move as well. This allows you to see the relationship between actions and the different model variables easily. This goes for the X/Y plots as well: if the X/T plots show a particular time range, the X/Y plots only show transitions from that time range.

F.7.2.4 Auto-adjustment of plots

It is not always handy to see all transitions at once. The simulator can automatically adjust time ranges for you during simulation. Drop-down box **16** controls which transitions are visible. It applies to the action viewer, X/T and X/Y plots. There are four choices:

Last transition Time ranges are only adjusted when the last transition does not fit in. In that case, the range is set to twice the length of the last transition.

All transitions The time range is adjusted to fit all transitions.

Fixed interval The time range is set to a specific length. The end of the time range coincides with the end of the last transition. The length is a user-specified setting (section F.8).

No adjustment Plots are not adjusted during simulation.

Each time you zoom or scroll a plot, the auto-adjustment option is automatically returned to "No adjustment".

F.8 Settings

There are several settings, all accessible by clicking **Tools**, **Options**. A dialog with three tabs appears. We list the settings per tab.

On the **Simulation** tab, you can find the following settings:

Lookahead time Sometimes, a flow transition can have any time length. The simulator cannot cope with this, so a maximum time length must be provided.

Max. recursion depth For models that are not substitutably guarded (definition 2.12), the user must specify how much effort to spend on calculating first transitions. This is achieved by setting a maximum on the number of times that the first transitions function may be calculated recursively for each recursion variable (see also section 4.1.2.3).

Max. solve duration In step-wise simulation, the simulator calculates the set of first transitions, and then starts to solve all transitions. If there are many transitions, this can take long. This setting is a maximum on the time spent solving. After this time, control is returned to the user to solve transitions manually.

Choose maximum time in random simulation When this box is checked, the maximum possible time will be chosen for each flow transition during random simulation. This can mean a huge performance improvement, since "half" flow transitions are usually not interesting.

Ignore zero-length flows during random simulation Checking this box prevents the simulator from choosing flow transitions that have a zero time length, when there is an alternative.

The **Mathematica** tab allows the user to enter the mathematica kernel to use. Usually, the kernel is located in
`C:\Program Files\Wolfram Research\Mathematica\5.1\MathKernel.exe`.

The **Visualization** tab shows the same options as drop-down box **16** (see section F.7.2.4), but also allows for entering the time interval for the "Fixed interval" option.

F.9 Building

If you want to build your own executable from the source code, you need JBuilder 2005, JavaCup 0.10, and Mathematica 5.1.

Open JBuilder, open the project "hypasim.jpx". First, you need to change some settings. Click **Project**, **Project properties...**, **Required libraries**. Click the "javacup" item and click **Edit...**. On the **Class** tab, enter the main directory of your JavaCUP installation. Do the same for the "mathematica" library, but enter the path to JLink.jar (supplied with Mathematica 5.1 in the AddOns\JLink directory).

Now you are ready to build the project. You can build Java class files using **Project**, **Rebuild project "hypasim.jpx"**. If you want to build Windows or Linux executables, right-click the native executable node in the project pane on the left (named "hypasim") and choose **Rebuild**.

Detailed implementation documentation can be generated by right-clicking the "Standard Doclet" node in the project pane, and choosing **Rebuild**. This generates JavaDoc HTML documentation.

Appendix G

Input file format grammar

For communication with future HyPA tools, it is convenient to have a precise description of the input file format. This appendix gives the grammars we used in combinations with the JFlex and JavaCUP tools to create a lexical scanner and a parser.

G.1 JFlex input

The following is the input we fed JFlex to create a lexical scanner. See the JFlex manual for details.

```
/* Input file for the JFlex lexical scanner generator */
/* Scanner definition for the HyPASim input file format number 1 */

package hypasim.engine.format1;

import hypasim.interface_engine_ui.*;
import java.util.*;
import java_cup.runtime.*;

%%

%class HyPA_ASCII_scanner
%unicode
%cupsym HyPA_ASCII_sym
%cup
%line
%column
%apiprivate
%switch
%yylexthrow EHSParseException

%{

  int bracketCount = 0;
  String predicate = "";

  private Symbol symbol(int type) {
    return new Symbol(type, yyline+1, yycolumn+1);
  }
```

```
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline+1, yycolumn+1, value);
  }

%}

LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace = {LineTerminator}|[ \t\f]
Comment = "%" {InputCharacter}*{LineTerminator}
Identifier = [:jletterdigit:]+
Predicate = [^\x5B\x5D]*

%state CLAUSEBEGIN
%state CLAUSEPRED
%state DEFINITIONBEGIN
%state DEFINITIONPRED

%%

<YYINITIAL> {

    /* headers */
    "clause formalism"  { return symbol(HyPA_ASCII_sym.CLAUSEFORMALISMHEADER); }
    "act"               { return symbol(HyPA_ASCII_sym.ACTHEADER); }
    "comm"              { return symbol(HyPA_ASCII_sym.COMMHEADER); }
    "var"               { return symbol(HyPA_ASCII_sym.VARHEADER); }
    "def"        { yybegin(DEFINITIONBEGIN); return symbol(HyPA_ASCII_sym.DEFHEADER); }
    "proc"              { return symbol(HyPA_ASCII_sym.PROCHEADER); }
    "initial"           { return symbol(HyPA_ASCII_sym.INITIALHEADER); }
    "actcomm"           { return symbol(HyPA_ASCII_sym.ACTCOMMHEADER); }

    /* constants */
    "delta"             { return symbol(HyPA_ASCII_sym.DELTA); }
    "epsilon"           { return symbol(HyPA_ASCII_sym.EPSILON); }

    /* comments and whitespace */
    {WhiteSpace}        { /* Ignore */ }
    {Comment}           { /* Ignore */ }

    /* separators */
    ","                 { return symbol(HyPA_ASCII_sym.COMMA); }
    "|["                { return symbol(HyPA_ASCII_sym.LABSTRACT); }
    "]|"                { return symbol(HyPA_ASCII_sym.RABSTRACT); }
    "|"                 { return symbol(HyPA_ASCII_sym.BAR); }
    "("                 { return symbol(HyPA_ASCII_sym.LPAREN); }
    ")"                 { return symbol(HyPA_ASCII_sym.RPAREN); }
    "["                 { yybegin(CLAUSEBEGIN); return symbol(HyPA_ASCII_sym.LSQBRACKET); }
    "{"                 { return symbol(HyPA_ASCII_sym.LBRACE); }
    "}"                 { return symbol(HyPA_ASCII_sym.RBRACE); }
    "="                 { return symbol(HyPA_ASCII_sym.EQUALS); }
    "->"                { return symbol(HyPA_ASCII_sym.RIGHTARROW); }
    "<->"               { return symbol(HyPA_ASCII_sym.LEFTRIGHTARROW); }
```

```
    /* operators */
    "~"                 { return symbol(HyPA_ASCII_sym.CONCAT); }
    "/\\"               { return symbol(HyPA_ASCII_sym.AND); }
    "\\/"               { return symbol(HyPA_ASCII_sym.OR); }
    "?"                 { return symbol(HyPA_ASCII_sym.SAT); }
    "cjmp"              { return symbol(HyPA_ASCII_sym.CJMP); }
    ">>"                { return symbol(HyPA_ASCII_sym.REINITOP); }
    "+"                 { return symbol(HyPA_ASCII_sym.ALTERNATIVEOP); }
    "."                 { return symbol(HyPA_ASCII_sym.SEQUENTIALOP); }
    "|>"                { return symbol(HyPA_ASCII_sym.DISRUPTOP); }
    "L>"                { return symbol(HyPA_ASCII_sym.LDISRUPTOP); }
    "||"                { return symbol(HyPA_ASCII_sym.PARALLELOP); }
    "L|"                { return symbol(HyPA_ASCII_sym.LPARALLELOP); }
    "encap"             { return symbol(HyPA_ASCII_sym.ENCAP); }
    "rename"            { return symbol(HyPA_ASCII_sym.ACTIONRENAME); }
    "varrename"         { return symbol(HyPA_ASCII_sym.VARRENAME); }

    /* identifiers */
    {Identifier}        { return symbol(HyPA_ASCII_sym.ID, yytext()); }

}

<CLAUSEBEGIN>{
    ","                 { return symbol(HyPA_ASCII_sym.COMMA); }
    {Identifier}        { return symbol(HyPA_ASCII_sym.ID, yytext()); }
    "|"
        {
            yybegin(CLAUSEPRED);
            predicate = "";
            bracketCount = 1;
            return symbol(HyPA_ASCII_sym.BAR);
        }
    {WhiteSpace}        { /* Ignore */ }
    {Comment}           { /* Ignore */ }
}

<CLAUSEPRED>{
    "["                 { bracketCount++; predicate += "[";}
    {Predicate}         { predicate += yytext(); }
    "]"
        { bracketCount--;
          if (bracketCount == 0){
            yybegin(YYINITIAL);
            return symbol(HyPA_ASCII_sym.PREDICATE, predicate);
          }
          else{
            predicate += "]";
          }
        }
}

<DEFINITIONBEGIN>{
    "["
        {
```

```
            yybegin(DEFINITIONPRED);
            predicate = "";
            bracketCount = 1;
            return symbol(HyPA_ASCII_sym.LSQBRACKET);
        }
    {WhiteSpace}        { /* Ignore */ }
    {Comment}           { /* Ignore */ }
}

<DEFINITIONPRED>{
    "["         { bracketCount++; predicate += "[";}
    {Predicate}         { predicate += yytext(); }
    "]"
        { bracketCount--;
          if (bracketCount == 0){
            yybegin(YYINITIAL);
            return symbol(HyPA_ASCII_sym.PREDICATE, predicate);
          }
          else{
            predicate += "]";
          }
        }
}

/* error fallback */
.|\n                     { throw new EHSParseException("Illegal character \"" +
                           yytext() + "\" encountered at line " + (yyline+1) + ",
                           column " + (yycolumn+1)); }
```

G.2 JavaCUP input

The following grammar leads JavaCUP to produce a parser for HyPA.

```
/* Input file for the CUP parser generator */
/* Parser definition for the HyPASim input file format number 1 */

package hypasim.engine.format1;

import java_cup.runtime.*;
import java.util.*;
import hypasim.interface_engine_ui.*;

action code {:
  ...
:}

parser code {:
  /* Constructor that creates a HyPA_ASCII_scanner */
  public HyPA_ASCII_parser (java.io.Reader input){
     ...
  }

  /* Parse method that returns a specification */
```

```
  public Symbol parse() throws java.lang.Exception{
    ...
  }

  /* Throw syntax error with line/col info */
  public void unrecovered_syntax_error(Symbol symbol) throws Exception {
      try{
          super.unrecovered_syntax_error(symbol);
      }
      catch (Exception e){
        throw new EHSParseException("Syntax error on line " +
            symbol.left + " column " + symbol.right);
      }
  }
:}

/* Terminals (tokens returned by the scanner). */
// headers
terminal CLAUSEFORMALISMHEADER, ACTHEADER, COMMHEADER, VARHEADER;
terminal DEFHEADER, PROCHEADER, INITIALHEADER, ACTCOMMHEADER;
// separators
terminal COMMA, BAR, EQUALS, LABSTRACT, RABSTRACT, LPAREN, RPAREN;
terminal LSQBRACKET, LBRACE, RBRACE, RIGHTARROW, LEFTRIGHTARROW;
// constants
terminal DELTA, EPSILON;
// operators
terminal CONCAT, AND, OR, CJMP, SAT, REINITOP, ALTERNATIVEOP, SEQUENTIALOP;
terminal DISRUPTOP, LDISRUPTOP, PARALLELOP, LPARALLELOP;
terminal ENCAP, ACTIONRENAME, VARRENAME;
// special
terminal String ID, PREDICATE;

/* Non terminals */
non terminal spec;
non terminal String section;
non terminal String clauseformalism, actions_section, var_section;
non terminal String action_list, var_list, var;
non terminal String comm_section, comm_list, comm;
non terminal String actcomm_section, actcomm_list, actcomm;
non terminal String definition_section;
non terminal String proc;
non terminal H1Element proc_expr;
non terminal TreeSet id_list;
non terminal TreeMap rename_list;

/* Precedences */
precedence left ALTERNATIVEOP;
precedence left BAR;
precedence left LPARALLELOP;
precedence left PARALLELOP;
precedence right REINITOP;
precedence left LDISRUPTOP;
precedence left DISRUPTOP;
precedence right SEQUENTIALOP;
```

```
precedence left AND, OR;
precedence left CONCAT;
precedence left CJMP, SAT;
precedence left LSQBRACKET;
precedence left LPAREN;

/* The grammar */
spec ::= section:s
    | spec:s1 section:s2
    ;

section ::= clauseformalism:s
     | actions_section:s
     | comm_section:s
     | actcomm_section:s
     | var_section:s
     | definition_section:s
     | proc:s
    ;

action_list ::= ID:s
    | action_list ID:s
    | action_list COMMA ID:s
    ;

clauseformalism ::= CLAUSEFORMALISMHEADER ID:i
    ;

actions_section ::= ACTHEADER action_list
    ;

comm_section ::=  COMMHEADER comm_list
    ;

definition_section ::= DEFHEADER LSQBRACKET PREDICATE:pred
    ;

actcomm_section ::=  ACTCOMMHEADER actcomm_list
    ;

comm_list ::= comm
    | comm_list comm
    | comm_list COMMA comm
    ;

comm ::= ID:s1 BAR ID:s2 EQUALS ID:s3
    ;

actcomm_list ::= actcomm
    | actcomm_list actcomm
    | actcomm_list COMMA actcomm
    ;

actcomm ::= ID:s1 BAR ID:s2 EQUALS ID:s3
```

```
    ;

var ::= ID:s
    {: ... :}
    ;

var_list ::= var
    | var_list var
    | var_list COMMA var
    ;

var_section ::= VARHEADER var_list
    ;

id_list ::= ID:s
    | id_list:l ID:s
    | id_list:l COMMA ID:s
    ;

rename_list ::= ID:i1 RIGHTARROW ID:i2
    | rename_list:l COMMA ID:i1 RIGHTARROW ID:i2
    | ID:i1 LEFTRIGHTARROW ID:i2
    | rename_list:l COMMA ID:i1 LEFTRIGHTARROW ID:i2
    ;

proc ::= PROCHEADER ID:var EQUALS proc_expr:p
    | INITIALHEADER PROCHEADER ID:var EQUALS proc_expr:p
    ;

proc_expr ::= ID:id
    | DELTA
    | EPSILON
    | LABSTRACT id_list:l BAR proc_expr:c1 RABSTRACT
    | LSQBRACKET id_list:l BAR PREDICATE:pred
    | LSQBRACKET BAR PREDICATE:pred
    | LSQBRACKET PREDICATE:pred
    | proc_expr:c1 AND proc_expr:c2
    | proc_expr:c1 OR proc_expr:c2
    | proc_expr:c1 CONCAT proc_expr:c2
    | proc_expr:d SAT
    | CJMP LPAREN proc_expr:c RPAREN
    | proc_expr:c REINITOP proc_expr:p
    | proc_expr:p ALTERNATIVEOP proc_expr:q
    | proc_expr:p SEQUENTIALOP proc_expr:q
    | proc_expr:p DISRUPTOP proc_expr:q
    | proc_expr:p LDISRUPTOP proc_expr:q
    | proc_expr:p PARALLELOP proc_expr:q
    | proc_expr:p LPARALLELOP proc_expr:q
    | proc_expr:p BAR proc_expr:q
    | ENCAP LPAREN LBRACE id_list:l RBRACE COMMA proc_expr:p RPAREN
    | ACTIONRENAME LPAREN LBRACE rename_list:l RBRACE COMMA proc_expr:p RPAREN
    | VARRENAME LPAREN LBRACE rename_list:l RBRACE COMMA proc_expr:p RPAREN
    | LPAREN proc_expr:p RPAREN
    ;
```